META-ADDRESS ARCHITECTURE FOR PARALLEL, DYNAMICALLY RECONFIGURABLE COMPUTING

5   ## CROSS-REFERENCE TO RELATED APPLICATIONS

The present invention is a continuation-in-part application of U.S. Patent Application Serial No. 09/031,323, entitled "SYSTEM AND METHOD FOR DYNAMICALLY RECONFIGURABLE COMPUTING USING A PROCESSING UNIT HAVING

10   CHANGEABLE INTERNAL HARDWARE ORGANIZATION," filed on February 26, 1998, which is a divisional application of U.S. Patent No. 5,794,062, filed on April 17, 1995.

BACKGROUND OF THE INVENTION

15
1.   Field of the Invention

The present invention relates generally to computer architecture, and more particularly to systems and methods for reconfigurable computing. Still more particularly, the present invention

20   is a system and method for scalable, parallel, dynamically reconfigurable computing.

2.   Description of the Background Art

The evolution of computer architecture is driven by the need for ever-greater

25   computational performance. Rapid, accurate solution of different types of computational problems typically requires different types of computational resources. For a given range of problem types, computational performance can be enhanced through the use of computational resources that have been specifically architected for the problem types under consideration. For example, the use of Digital Signal Processing (DSP) hardware in conjunction with a general-

30   purpose computer can significantly enhance certain types of signal processing performance. In the event that a computer itself has been specifically architected for the problem types under consideration, computational performance will be further enhanced, or possibly even optimized relative to the available computational resources, for these particular problem types. Current parallel and massively-parallel computers, offering high performance for specific types of

35   problems of $O(n^2)$ or greater complexity, provide examples in this case.

The need for greater computational performance must be balanced against the need to minimize system cost and the need to maximize system productivity in a widest-possible range of both current-day and possible future applications. In general, the incorporation of computational resources dedicated to a limited number of problem types into a computer system

40   adversely affects system cost because specialized hardware is typically more expensive than

general-purpose hardware. The design and production of an entire special-purpose computer can be prohibitively expensive in terms of both engineering time and hardware costs. The use of dedicated hardware to increase computational performance may offer few performance benefits as computational needs change. In the prior art, as computational needs have changed, new types

5   of specialized hardware or new special-purpose systems have been designed and manufactured, resulting in an ongoing cycle of undesirably large nonrecurrent engineering costs. The use of computational resources dedicated to particular problem types therefore results in an inefficient use of available system Silicon when considering changing computational needs. Thus, for the reasons described above, attempting to increase computational performance using dedicated

10   hardware is undesirable.

        In the prior art, various attempts have been made to both increase computational performance and maximize problem type applicability using reprogrammable or reconfigurable hardware. A first such prior art approach is that of downloadable microcode computer architectures. In a downloadable microcode architecture, the behavior of fixed,

15   nonreconfigurable hardware resources can be selectively altered by using a particular version of microcode. An example of such an architecture is that of the IBM System/360. Because the fundamental computational hardware in such prior art systems is not itself reconfigurable, such systems do not provide optimized computational performance when considering a wide range of problem types.

20   A second prior art approach toward both increasing computational performance and maximizing problem type applicability is the use of reconfigurable hardware coupled to a nonreconfigurable host processor or host system. This prior art approach most commonly involves the use of one or more reconfigurable co-processors coupled to a nonreconfigurable host. This approach can be categorized as an "Attached Reconfigurable Processor" (ARP)

25   architecture, where some portion of hardware within a processor set attached to a host is reconfigurable. Examples of present-day ARP systems that utilize a set of reconfigurable processors coupled to a host system include: the SPLASH-1 and SPLASH-2 systems, designed at the Supercomputing Research Center (Bowie, MD); the WILDFIRE Custom Configurable Computer produced by Annapolis Micro Systems (Annapolis, MD), which is a commercial

30   version of the SPLASH-2; and the EVC-1, produced by the Virtual Computer Corporation (Reseda, CA). In most computation-intensive problems, significant amounts of time are spent executing relatively small portions of program code. In general, ARP architectures are used to provide a reconfigurable computational accelerator for such portions of program code. Unfortunately, a computational model based upon one or more reconfigurable computational

35   accelerators suffers from significant drawbacks, as will be described in detail below.

A first drawback of ARP architectures arises because ARP systems attempt to provide an optimized implementation of a particular algorithm in reconfigurable hardware at a particular time. The philosophy behind Virtual Computer Corporation's EVC-1, for example, is the conversion of a specific algorithm into a specific configuration of reconfigurable hardware

5   resources to provide optimized computational performance for that particular algorithm. Reconfigurable hardware resources are used for the sole purpose of providing optimum performance for a specific algorithm. The use of reconfigurable hardware resources for more general purposes, such as managing instruction execution, is avoided. Thus, for a given algorithm, reconfigurable hardware resources are considered from the perspective of individual

10   gates coupled to ensure optimum performance.

Certain ARP systems rely upon a programming model in which a "program" includes both conventional program instructions as well as special-purpose instructions that specify how various reconfigurable hardware resources are interconnected. Because ARP systems consider reconfigurable hardware resources in a gate-level algorithm-specific manner, these special-

15   purpose instructions must provide explicit detail as to the nature of each reconfigurable hardware resource used and the manner in which it is coupled to other reconfigurable hardware resources. This adversely affects program complexity. To reduce program complexity, attempts have been made to utilize a programming model in which a program includes both conventional high-level programming language instructions as well as high-level special-purpose instructions. Current

20   ARP systems therefore attempt to utilize a compiling system capable of compiling both high-level programming language instructions and the aforementioned high-level special-purpose instructions. The target output of such a compiling system is assembly-language code for the conventional high-level programming language instructions, and Hardware Description Language (HDL) code for the special-purpose instructions. Unfortunately, the automatic

25   determination of a set of reconfigurable hardware resources and an interconnection scheme to provide optimal computational performance for any particular algorithm under consideration is an NP-hard problem. A long-term goal of some ARP systems is the development of a compiling system that can compile an algorithm directly into an optimized interconnection scheme for a set of gates. The development of such a compiling system, however, is an exceedingly difficult task,

30   particularly when considering multiple types of algorithms.

A second shortcoming of ARP architectures arises because an ARP apparatus distributes the computational work associated with the algorithm for which it is configured across multiple reconfigurable logic devices. For example, for an ARP apparatus implemented using a set of Field Programmable Logic Devices (FPGAs) and configured to implement a parallel

35   multiplication accelerator, the computational work associated with parallel multiplication is distributed across the entire set of FPGAs. Therefore, the size of the algorithm for which the

ARP apparatus can be configured is limited by the number of reconfigurable logic devices present. The maximum data-set size that the ARP apparatus can handle is similarly limited. An examination of source code does not necessarily provide a clear indication of the limitations of the ARP apparatus because some algorithms may have data dependencies. In general, data-

5   dependent algorithms are avoided.

Furthermore, because ARP architectures teach the distribution of computational work across multiple reconfigurable logic devices, accommodation of a new (or even slightly modified) algorithm requires that reconfiguration be done en masse, that is, multiple reconfigurable logic devices must be reconfigured. This limits the maximum rate at which

10   reconfiguration can occur for alternative problems or cascaded subproblems.

A third drawback of ARP architectures arises from the fact that one or more portions of program code are executed on the host. That is, an ARP apparatus is not an independent computing system in itself, the ARP apparatus does not execute entire programs, and therefore interaction with the host is required. Because some program code is executed upon the

15   nonreconfigurable host, the set of available Silicon resources is not maximally utilized over the time-frame of the program's execution. In particular, during host-based instruction execution, Silicon resources upon the ARP apparatus will be idle or inefficiently utilized. Similarly, when the ARP apparatus operates upon data, Silicon resources upon the host will, in general, be inefficiently utilized. In order to readily execute multiple entire programs, Silicon resources

20   within a system must be grouped into readily reusable resources. As previously described, ARP systems treat reconfigurable hardware resources as a set of gates optimally interconnected for the implementation of a particular algorithm at a particular time. Thus, ARP systems do not provide a means for treating a particular set of reconfigurable hardware resources as a readily reusable resource from one algorithm to another because reusability requires a certain level of algorithmic

25   independence.

An ARP apparatus cannot treat its currently-executing host program as data, and in general cannot contextualize itself. An ARP apparatus could not readily be made to simulate itself through the execution of its own host programs. Furthermore, an ARP apparatus could not be made to compile its own HDL or application programs upon itself, directly using the

30   reconfigurable hardware resources from which it is constructed. An ARP apparatus is thus architecturally limited in relation to self-contained computing models that teach independence from a host processor.

Because an ARP apparatus functions as a computational accelerator, it in general is not capable of independent Input/Output (I/O) processing. Typically, an ARP apparatus requires

35   host interaction for I/O processing. The performance of an ARP apparatus may therefore be I/O limited. Those skilled in the art will recognize that an ARP apparatus can, however, be

configured for accelerating a specific I/O problem. However, because the entire ARP apparatus is configured for a single, specific problem, an ARP apparatus cannot balance I/O processing with data processing without compromising one or the other. Moreover, an ARP apparatus provides no means for interrupt processing. ARP teachings offer no such mechanism because

5   they are directed toward maximizing computational acceleration, and interruption negatively impacts computational acceleration.

A fourth drawback of ARP architectures exists because there are software applications that possess inherent data parallelism that is difficult to exploit using an ARP apparatus. HDL compilation applications provide one such example when net-name symbol resolution in a very

10   large netlist is required.

A fifth drawback associated with ARP architectures is that they are essentially a SIMD computer architecture model. ARP architectures are therefore less effective architecturally than one or more innovative prior art nonreconfigurable systems. ARP systems mirror only a portion of the process of executing a program, chiefly, the arithmetic logic for arithmetic computation,

15   for each specific configuration instance, for as much computational power as the available reconfigurable hardware can provide. In contradistinction, in the system design of the SYMBOL machine at Fairchild in 1971, the entire computer used a unique hardware context for every aspect of program execution. As a result, SYMBOL encompassed every element for the system application of a computer, including the host portion taught by ARP systems.

20   ARP architectures exhibit other shortcomings as well. For example, an ARP apparatus lacks an effective means for providing independent timing to multiple reconfigurable logic devices. Similarly, cascaded ARP apparatus lack an effective clock distribution means for providing independently-timed units. As another example, it is difficult to accurately correlate execution time with the source code statements for which acceleration is attempted. For an

25   accurate estimate of net system clock rate, the ARP device must be modeled with a Computer-Aided Design (CAD) tool after HDL compilation, a time-consuming process for arriving at such a basic parameter.

An equally significant problem with conventional architectures is their use of virtual or shared memory. This teaching of using a unified address space results in slower, less efficient

30   memory access due to the more complicated addressing operations required. For example, in order to access individual bits in the memory device of a system using virtual memory, the physical address space of the memory must be first segmented into logical addresses, and then virtual addresses must be mapped onto the logical addresses. Only then may the bits in the memory be accessed. Additionally, in shared memory systems the processor typically performs

35   address validation operations prior to allowing access to the memory, further complicating the memory operation. Finally, the processor must arbitrate between multiple processes attempting

to access the same area of memory at the same time by providing some type of prioritization system.

To address the myriad of problems caused by the use of shared and virtual memory, many conventional systems use memory management units (MMUs) to perform the majority of the

5   memory management functions, such as converting logical addresses to virtual addresses. However, the MMU/software interaction adds yet another degree of complexity to the memory accessing operation. Additionally, MMUs are quite limited in the types of operations which they can perform. They cannot handle interrupts, queue messages, or perform sophisticated addressing operations which all must be performed by the processor. When shared or virtual

10   memory systems are employed in a computer architecture which has multiple parallel processors, the above-described defects are magnified. Not only must the hardware/software interactions be managed as described above, but the coherence and consistency of the data in the memory must also be maintained by both software <u>and</u> hardware in response to multiple processors attempting to access the shared memory. The addition of more processors increases the difficulty of the

15   virtual address to logical address conversion. These complications in the memory accessing operation necessarily degrade system performance; this degradation only increases as the system grows larger as more processors are added.

One example of a conventional system is the cache-coherent, Non-Uniform Memory Access (ccNUMA) computer architecture. The ccNUMA machines use complex and costly

20   hardware, such as cache controllers and crossbar switches, to maintain for each independent CPU the illusion of a single address space even though the memory is actually shared by multiple processors. The ccNUMA is moderately scalable, but achieves this scalability by the use of the additional hardware to achieve tight coupling of the processors in its system. This type of system is more advantageously used in computing environment in which a single program image is

25   being shared, where shared memory I/O operations have very large bandwidth requirements, such as for finite element grids in scientific computing. Further, the ccNUMA is not useful for systems in which processors are not similar in nature. The ccNUMA architecture requires that each processor added be of the same type as the existing processors. In a system in which processors are optimized to serve different functions, and therefore operate differently from each

30   other, the ccNUMA architecture does not provide an effective solution. Finally, in conventional systems, only the standard memory addressing schemes are used to address memory in the system.

What is needed is a means for addressing memory in a parallel computing environment which provides for scalability, transparent addressing, and which has a minimal impact on the

35   processing power of the system.

SUMMARY OF THE INVENTION

The present invention is a system and method for scalable, parallel, dynamically reconfigurable computing. The system comprises at least one S-machine, a T-machine corresponding to each S-machine, a General-Purpose Interconnect Matrix (GPIM), a set of I/O T-machines, one or more I/O devices, and a master time-base unit. In the preferred embodiment, the system includes multiple S-machines. Each S-machine has an input and an output coupled to an output and an input of a corresponding T-machine, respectively. Each T-machine includes a routing input and a routing output coupled to the GPIM, as does each I/O T-machine. An I/O T-machine further includes an input and an output coupled to an I/O device. Finally, each S-machine, T-machine, and I/O T-machine has a master timing input coupled to a timing output of the master time-base unit.

The meta-addressing system of the present invention provides for bit-addressable capabilities for the processors in the network without requiring the processors themselves to perform the processing-intensive address manipulation functions. Separate processing and addressing machines are disclosed which are optimized to perform their assigned functions. The processing machines execute instructions, store and retrieve data from a local memory, and determine when remote operations are required. The addressing machines assemble packets of data for transmission, determine a geographic or network address of the packet, and perform addressing checking on incoming packets. Additionally, the addressing machines can provide interrupt handling and other addressing operations.

In one embodiment, the T-machines also provide the meta-addressing mechanism of the present invention. The meta-addresses designate the geographic location of the T-machines in the system and specify the location of data within the local memory devices. The local address of the meta-address is be used to address each bit in the memory of the new device, regardless of the actual memory size of the device (as long as the addressable space of the device is less or equal to the bit count of the local address). Thus, devices having different memory sizes and structures may be addressed using the single meta-address. Further, by use of the meta-address, hardware within the multi-processor parallel architecture is not required to guarantee coherency and consistency across the system.

The meta-address allows for complete scalability; as a new S-machine or I/O device is added, a new geographic address is designated for the new device. The present invention allows for irregular scalability, in that there is no requirement of a power-of-two number of processors. Scalability is also enhanced by the ability to couple any number of addressing machines to each processing machines, up to the available local memory bandwidth. This allows the system

designer to arbitrarily designate the number of pathways to each processing machine. This flexibility can be used to allow more communication bandwidth to be provided to higher levels of the system, creating in effect a pyramid processing architecture which is optimized to devote the most communication bandwidth to the most important functions of the system.

5   As described above, in accordance with a preferred embodiment, the T-machines are addressing machines which generate meta-addresses, handle interrupts, and queue messages. The S-machines are thus freed to devote their processing capacity solely on the execution of program instructions, greatly optimizing the overall efficacy of the multi-processor parallel architecture of the present invention. The S-machines need only access the local memory

10   component of the meta-address to locate the desired data; the geographic address is transparent to the S-machine. This addressing architecture interoperates extremely well with a distributed memory/distributed processor parallel computing system. The architectural design choice of isolating the local memories allows independent and parallel operation of hardware. In accordance with the present invention, each S-machine can have completely divergent

15   reconfiguration directives at runtime, even though all are directed in parallel on one computing problem. Also, not only can the Instruction Set Architectures realized by dynamically reconfigurable S-machines be different, the actual hardware used to realize the S-machines can be optimized to perform certain tasks. Thus, the S-machines in a single system may all be operating at different rates, allowing each S-machine to optimally perform its function while

20   maximizing the use of system resources.

       Additionally, the only memory validation which occurs is to verify the correct geographic address has been transmitted; no validation of the local memory address is provided. Further, this validation is performed by the addressing machine, not by the processing machine. As no virtual addressing is used, no hardware/software interoperations for converting virtual addresses

25   to logical addresses is required. The address in the meta-address is the physical address. The elimination of all of these preventative and maintenance functions greatly increases the processing speed of the entire system. Thus, by separating the "space" management of computer systems into separate addressing machines from the "time" management of the computer system (provided by the separate processing machines), in combination with the meta-addressing

30   scheme, a unique memory management and addressing system for highly parallel computing systems is provided. The architecture of the present invention allows great flexibility in the operations of the S-machines, allowing each S-machine to operate at its own optimal rate, while maintaining a uniform T-machine rate. This balance of local instruction processing in fastest time, with system-wide data communication provided for across the farthest space, provides an

35   improved approach to complex problem solving by highly parallel computer systems.

BRIEF DESCRIPTION OF THE DRAWINGS

5   Figure 1 is a block diagram of a preferred embodiment of a system for scalable, parallel, dynamically reconfigurable computing constructed in accordance with the present invention;

      Figure 2 is a block diagram of a preferred embodiment of an S-machine of the present invention;

10

      Figure 3A is an exemplary program listing that includes reconfiguration directives;

      Figure 3B is a flowchart of prior art compiling operations performed during the compilation of a sequence of program instructions;

15

      Figures 3C and 3D are a flowchart of preferred compiling operations performed by a compiler for dynamically reconfigurable computing;

      Figure 4 is a block diagram of a preferred embodiment of a Dynamically Reconfigurable

20   Processing Unit of the present invention;

      Figure 5 is a block diagram of a preferred embodiment of an Instruction Fetch Unit of the present invention;

25   Figure 6 is a state diagram showing a preferred set of states supported by an Instruction State Sequencer of the present invention;

      Figure 7 is a state diagram showing a preferred set of states supported by interrupt logic of the present invention;

30

      Figure 8 is a block diagram of a preferred embodiment of a Data Operate Unit of the present invention;

      Figure 9A is a block diagram of a first exemplary embodiment of the Data Operate Unit

35   configured for the implementation of a general-purpose outer-loop Instruction Set Architecture;

Figure 9B is a block diagram of a second exemplary embodiment of the Data Operate Unit configured for the implementation of an inner-loop Instruction Set Architecture;

Figure 10 is a block diagram of a preferred embodiment of an Address Operate Unit of the present invention;

5

Figure 11A is a block diagram of a first exemplary embodiment of the Address Operate Unit configured for the implementation of a general-purpose outer-loop Instruction Set Architecture;

10

Figure 11B is a block diagram of a second exemplary embodiment of the Address Operate Unit configured for the implementation of an inner-loop Instruction Set Architecture;

Figure 12A is a diagram showing an exemplary allocation of reconfigurable hardware resources between the Instruction Fetch Unit, the Data Operate Unit, and the Address Operate Unit for an outer-loop Instruction Set Architecture;

15

Figure 12B is a diagram showing an exemplary allocation of reconfigurable hardware resources between the Instruction Fetch Unit, the Data Operate Unit, and the Address Operate Unit for an inner-loop Instruction Set Architecture;

20

Figure 13 is a block diagram of a preferred embodiment of a T-machine of the present invention;

25

Figure 14 is a block diagram of an interconnect I/O unit of the present invention;

Figure 15 is a block diagram of a preferred embodiment of an I/O T-machine of the present invention;

30

Figure 16 is a block diagram of a preferred embodiment of a General Purpose Interconnect Matrix of the present invention; and

Figures 17A and 17B are a flowchart of a preferred method for scalable, parallel, dynamically reconfigurable computing in accordance with the present invention.

35

Figure 18 is a block diagram of a preferred embodiment of a data packet in accordance with the present invention.

Figure 19 is a flow chart of a preferred method for generating a data request in
accordance with the present invention.

Figure 20 is a flow chart of a preferred method for sending data in accordance with the present invention.

Figure 21 is a flow chart of a preferred method for receiving data in accordance with the present invention.

Figure 22 is a block diagram of a preferred embodiment of the interconnect I/O unit which performs interrupt handling operations in accordance with the present invention.

Figure 23 is a flow chart of a preferred method for handling interrupts in accordance with the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Referring now to Figure 1, a block diagram of a preferred embodiment of a system 10 for scalable, parallel, dynamically reconfigurable computing constructed in accordance with the present invention is shown. The system 10 preferably comprises at least one S-machine 12, a T-machine 14 corresponding to each S-machine 12, a General Purpose Interconnect Matrix (GPIM) 16, at least one I/O T-machine 18, one or more I/O devices 20, and a master time-base unit 22. In the preferred embodiment, the system 10 comprises multiple S-machines 12, and thus multiple T-machines 14, plus multiple I/O T-machines 18 and multiple I/O devices 20.

Each of the S-machines 12, T-machines 14, and I/O T-machines 18 has a master timing input coupled to a timing output of the master time-base unit 22. Each S-machine 12 has an input and an output coupled to its corresponding T-machine 14. In addition to the input and the output coupled to its corresponding S-machine 12, each T-machine 14 has a routing input and a routing output coupled to the GPIM 16. In a similar manner, each I/O T-machine 18 has an input and an output coupled to an I/O device 20, and a routing input and a routing output to the GPIM 16.

As will be described in detail below, each S-machine 12 is a dynamically-reconfigurable computer. The GPIM 16 forms a point-to-point parallel interconnect means that facilitates communication between T-machines 14. The set of T-machines 14 and the GPIM 16 form a point-to-point parallel interconnect means for data transfer between S-machines 12. Similarly,

5 the GPIM 16, the set of T-machines 14, and the set of I/O T-machines 18 form a point-to-point parallel interconnect means for I/O transfer between S-machines 12 and each I/O device 20. The master time-base unit 22 comprises an oscillator that provides a master timing signal to each S-machine 12 and T-machine 14.

In an exemplary embodiment, each S-machine 12 is implemented using a Xilinx XC4013

10 (Xilinx, Inc., San Jose, CA) Field Programmable Gate Array (FPGA) coupled to 64 Megabytes of Random Access Memory (RAM). Each T-machine 14 is implemented using approximately fifty percent of the reconfigurable hardware resources in a Xilinx XC4013 FPGA, as is each I/O T-machine 18. The GPIM 14 is implemented as a toroidal interconnect mesh. The master time-base unit 22 is a clock oscillator coupled to clock distribution circuitry to provide a system-wide

15 frequency reference, as described in U.S. Patent Application Serial No. _____, entitled "System and Method for Phase-Synchronous, Flexible Frequency Clocking and Messaging." Preferably, the GPIM 14, the T-machines 12, and the I/O T-machines 18 transfer information in accordance with ANSI/IEEE Standard 1596-1992 defining a Scalable Coherent Interface (SCI).

20 In the preferred embodiment, the system 10 comprises multiple S-machines 12 functioning in parallel. The structure and functionality of each individual S-machine 12 are described in detail below with reference to Figures 2 through 12B. Referring now to Figure 2, a block diagram of a preferred embodiment of an S-machine 12 is shown. The S-machine 12 comprises a first local time-base unit 30, a Dynamically Reconfigurable Processing Unit (DRPU)

25 32 for executing program instructions, and a memory 34. The first local time-base unit 30 has a timing input that forms the S-machine's master timing input. The first local time-base unit 30 also has a timing output that provides a first local timing signal or clock to a timing input of the DRPU 32 and a timing input of the memory 34 via a first timing signal line 40. The DRPU 32 has a control signal output coupled to a control signal input of the memory 34 via a memory

30 control line 42; an address output coupled to an address input of the memory 34 via an address line 44; and a bidirectional data port coupled to a bidirectional data port of the memory 34 via a memory I/O line 46. The DRPU 32 additionally has a bidirectional control port coupled to a bidirectional control port of its corresponding T-machine 14 via an external control line 48. As shown in Figure 2, the memory control line 42 spans X bits, the address line 44 spans M bits, the

35 memory I/O line 46 spans (N x k) bits, and the external control line 48 spans Y bits.

In the preferred embodiment, the first local time-base unit 30 receives the master timing signal from the master time-base unit 22. The first local time-base unit 30 generates the first local timing signal from the master timing signal, and delivers the first local timing signal to the DRPU 32 and the memory 34. In the preferred embodiment, the first local timing signal can

5   vary from one S-machine 12 to another. Thus, the DRPU 32 and the memory 34 within a given S-machine 12 function at an independent clock rate relative to the DRPU 32 and the memory 34 within any other S-machine 12. Preferably, the first local timing signal is phase-synchronized with the master timing signal. In the preferred embodiment, the first local time-base unit 30 is implemented using phase-locked frequency-conversion circuitry, including phase-lock detection

10   circuitry implemented using reconfigurable hardware resources. Those skilled in the art will recognize that in an alternate embodiment, the first local time-base unit 30 could be implemented as a portion of a clock distribution tree.

The memory 34 is preferably implemented as a RAM, and stores program instructions, program data, and configuration data sets for the DRPU 32. The memory 34 of any given S-

15   machine 12 is preferably accessible to any other S-machine 12 in the system 10 via the GPIM 16. Moreover, each S-machine 12 is preferably characterized as having a uniform memory address space. In the preferred embodiment, program instructions stored in the memory 34 selectively include reconfiguration directives directed toward the DRPU 32. Referring now to Figure 3A, an exemplary program listing 50 including reconfiguration directives is shown. As shown in Figure

20   3A, the exemplary program listing 50 includes a set of outer-loop portions 52, a first inner-loop portion 54, a second inner-loop portion 55, a third inner-loop portion 56, a fourth inner-loop portion 57, and a fifth inner loop portion 58. Those skilled in the art will readily recognize that the term "inner-loop" refers to an iterative portion of a program that is responsible for performing a particular set of related operations, and the term "outer-loop" refers to those portions of a

25   program that are mainly responsible for performing general-purpose operations and/or transferring control from one inner-loop portion to another. In general, inner-loop portions 54, 55, 56, 57, 58 of a program perform specific operations upon potentially large data sets. In an image processing application, for example, the first inner-loop portion 54 might perform color-format conversion operations upon image data, and the second through fifth inner-loop portions

30   55, 56, 57, 58 might perform linear filtering, convolution, pattern searching, and compression operations. Those skilled in the art will recognize that a contiguous sequence of inner-loop portions 55, 56, 57, 58 can be thought of as a software pipeline. Each outer-loop portion 52 would be responsible for data I/O and/or directing the transfer of data and control from the first inner-loop portion 54 to the second inner-loop portion 55. Those skilled in the art will

35   additionally recognize that a given inner-loop portion 54, 55, 56, 57, 58 may include one or more reconfiguration directives. In general, for any given program, the outer-loop portions 52 of the

program listing 50 will include a variety of general-purpose instruction types, while the inner-loop portions 54, 56 of the program listing 50 will consist of relatively few instruction types used to perform a specific set of operations.

In the exemplary program listing 50, a first reconfiguration directive appears at the beginning of the first inner-loop portion 54, and a second reconfiguration directive appears at the end of the first inner-loop portion 54. Similarly, a third reconfiguration directive appears at the beginning of the second inner-loop portion 55; a fourth reconfiguration directive appears at the beginning of the third inner-loop portion 56; a fifth reconfiguration directive appears at the beginning of the fourth inner-loop portion 57; and a sixth and seventh reconfiguration directive appear at the beginning and end of the fifth inner-loop portion 58, respectively. Each reconfiguration directive preferably references a configuration data set that specifies an internal DRPU hardware organization dedicated to and optimized for the implementation of a particular Instruction Set Architecture (ISA). An ISA is a primitive or core set of instructions that can be used to program a computer. An ISA defines instruction formats, opcodes, data formats, addressing modes, execution control flags, and program-accessible registers. Those skilled in the art will recognize that this corresponds to the conventional definition of an ISA. In the present invention, each S-machine's DRPU 32 can be rapidly runtime-configured to directly implement multiple ISAs through the use of a unique configuration data set for each desired ISA. That is, each ISA is implemented with a unique internal DRPU hardware organization as specified by a corresponding configuration data set. Thus, in the present invention, the first through fifth inner-loop portions 54, 55, 56, 57, 58 each correspond to a unique ISA, namely, ISA 1, 2, 3, 4, and k, respectively. Those skilled in the art will recognize that each successive ISA need not be unique. Thus, ISA k could be ISA 1, 2, 3, 4, or any different ISA. The set of outer loop portions 52 also corresponds to a unique ISA, namely, ISA 0. In the preferred embodiment, during program execution the selection of successive reconfiguration directives may be data-dependent. Upon selection of a given reconfiguration directive, program instructions are subsequently executed according to a corresponding ISA via a unique DRPU hardware configuration as specified by a corresponding configuration data set.

In the present invention, a given ISA can be categorized as an inner-loop ISA or an outer-loop ISA according to the number and types of instructions it contains. An ISA that includes several instructions and that is useful for performing general-purpose operations is an outer-loop ISA, while an ISA that consists of relatively few instructions and that is directed to performing specific types of operations is an inner-loop ISA. Because an outer-loop ISA is directed to performing general-purpose operations, an outer-loop ISA is most useful when sequential execution of program instructions is desirable. The execution performance of an outer-loop ISA is preferably characterized in terms of clock cycles per instruction executed. In contrast, because

an inner-loop ISA is directed to performing specific types of operations, an inner-loop ISA is most useful when parallel program instruction execution is desirable. The execution performance of an inner-loop ISA is preferably characterized in terms of instructions executed per clock cycle or computational results produced per clock cycle.

5   Those skilled in the art will recognize that the preceding discussion of sequential program instruction execution and parallel program instruction execution pertains to program instruction execution within a single DRPU 32. The presence of multiple S-machines 12 in the system 10 facilitates the parallel execution of multiple program instruction sequences at any given time, where each program instruction sequence is executed by a given DRPU 32. Each DRPU 32 is

10   configured to have parallel or serial hardware to implement a particular inner-loop ISA or outer-loop ISA, respectively, at a particular time. The internal hardware configuration of any given DRPU 32 changes with time according to the selection of one or more reconfiguration directives embedded within a sequence of program instructions being executed.

In the preferred embodiment, each ISA and its corresponding internal DRPU hardware

15   organization are designed to provide optimum computational performance for a particular class of computational problems relative to a set of available reconfigurable hardware resources. As previously mentioned and as will be described in further detail below, an internal DRPU hardware organization corresponding to an outer-loop ISA is preferably optimized for sequential program instruction execution, and an internal DRPU hardware organization corresponding to an

20   inner-loop ISA is preferably optimized for parallel program instruction execution. An exemplary general-purpose outer-loop ISA is given in Appendix A, and an exemplary inner-loop ISA dedicated to convolution is given in Appendix B.

With the exception of each reconfiguration directive, the exemplary program listing 50 of Figure 3A preferably comprises conventional high-level language statements, for example,

25   statements written in accordance with the C programming language. Those skilled in the art will recognize that the inclusion of one or more reconfiguration directives in a sequence of program instructions requires a compiler modified to account for the reconfiguration directives. Referring now to Figure 3B, a flowchart of prior art compiling operations performed during the compilation of a sequence of program instructions is shown. Herein, the prior art compiling

30   operations correspond in general to those performed by the GNU C Compiler (GCC) produced by the Free Software Foundation (Cambridge, MA). Those skilled in the art will recognize that the prior art compiling operations described below can be readily generalized for other compilers. The prior art compiling operations begin in step 500 with the compiler front-end selecting a next high-level statement from a sequence of program instructions. Next, the

35   compiler front-end generates intermediate-level code corresponding to the selected high-level statement in step 502, which in the case of GCC corresponds to Register Transfer Level (RTL)

statements. Following step 502, the compiler front-end determines whether another high-level statement requires consideration in step 504. If so, the preferred method returns to step 500.

If in step 504 the compiler front-end determines that no other high-level statement requires consideration, the compiler back-end next performs conventional register allocation operations in step 506. After step 506, the compiler back-end selects a next RTL statement for consideration within a current RTL statement group in step 508. The compiler back-end then determines whether a rule specifying a manner in which the current RTL statement group can be translated into a set of assembly-language statements exists in step 510. If such a rule does not exist, the preferred method returns to step 508 to select another RTL statement for inclusion in the current RTL statement group. If a rule corresponding to the current RTL statement group exists, the compiler back-end generates a set of assembly-language statements according to the rule in step 512. Following step 512, the compiler back-end determines whether a next RTL statement requires consideration, in the context of a next RTL statement group. If so, the preferred method returns to step 508; otherwise, the preferred method ends.

The present invention preferably includes a compiler for dynamically reconfigurable computing. Referring also now to Figures 3C and 3D, a flowchart of preferred compiling operations performed by a compiler for dynamically reconfigurable computing is shown. The preferred compiling operations begin in step 600 with the front-end of the compiler for dynamically reconfigurable computing selecting a next high-level statement within a sequence of program instructions. Next, the front-end of the compiler for dynamically reconfigurable computing determines whether the selected high-level statement is a reconfiguration directive in step 602. If so, the front-end of the compiler for dynamically reconfigurable computing generates an RTL reconfiguration statement in step 604, after which the preferred method returns to step 600. In the preferred embodiment, the RTL reconfiguration statement is a non-standard RTL statement that includes an ISA identification. If in step 602 the selected high-level program statement is a not a reconfiguration directive, the front-end of the compiler for dynamically reconfigurable computing next generates a set of RTL statements in a conventional manner in step 606. After step 606, the front-end of the compiler for dynamically reconfigurable computing determines whether another high-level statement requires consideration in step 608. If so, the preferred method returns to step 600; otherwise, the preferred method proceeds to step 610 to initiate back-end operations.

In step 610, the back-end of the compiler for dynamically reconfigurable computing performs register allocation operations. In the preferred embodiment of the present invention, each ISA is defined such that the register architecture from one ISA to another is consistent; therefore, the register allocation operations are performed in a conventional manner. Those skilled in the art will recognize that in general, a consistent register architecture from one ISA to

another is not an absolute requirement. Next, the back-end of the compiler for dynamically reconfigurable computing selects a next RTL statement within a currently-considered RTL statement group in step 612. The back-end of the compiler for dynamically reconfigurable computing then determines in step 614 whether the selected RTL statement is an RTL

5　reconfiguration statement. If the selected RTL statement is not an RTL reconfiguration statement, the back-end of the compiler for dynamically reconfigurable computing determines in step 618 whether a rule exists for the currently-considered RTL statement group. If not, the preferred method returns to step 612 to select a next RTL statement for inclusion in the currently-considered RTL statement group. In the event that a rule exists for the currently-considered RTL

10　statement group in step 618, the back end of the compiler for dynamically reconfigurable computing next generates a set of assembly language statements corresponding to the currently-considered RTL statement group according to this rule in step 620. Following step 620, the back end of the compiler for dynamically reconfigurable computing determines whether another RTL statement requires consideration within the context of a next RTL statement group in step 622. If

15　so, the preferred method returns to step 612; otherwise, the preferred method ends.

If in step 614 the selected RTL statement is an RTL reconfiguration statement, the back-end of the compiler for dynamically reconfigurable computing selects a rule-set corresponding to the ISA identification within the RTL reconfiguration statement in step 616. In the present invention, a unique rule-set preferably exists for each ISA. Each rule-set therefore provides one

20　or more rules for converting groups of RTL statements into assembly language statements in accordance with a particular ISA. Following step 616, the preferred method proceeds to step 618. The rule set corresponding to any given ISA preferably includes a rule for translating the RTL reconfiguration statement into a set of assembly language instructions that produce a software interrupt that results in the execution of a reconfiguration handler, as will be described

25　in detail below.

In the manner described above, the compiler for dynamically reconfigurable computing selectively and automatically generates assembly-language statements in accordance with multiple ISAs during compilation operations. In other words, during the compilation process, the compiler for dynamically reconfigurable computing compiles a single set of program

30　instructions according to a variable ISA. The compiler for dynamically reconfigurable computing is preferably a conventional compiler modified to perform the preferred compiling operations described above with reference to Figures 3C and 3D. Those skilled in the art will recognize that while the required modifications are not complex, such modifications are nonobvious in view of both prior art compiling techniques and prior art reconfigurable

35　computing techniques.

Referring now to Figure 4, a block diagram of a preferred embodiment of a Dynamically Reconfigurable Processing Unit 32 is shown. The DRPU 32 comprises an Instruction Fetch Unit (IFU) 60, a Data Operate Unit (DOU) 62, and an Address Operate Unit (AOU) 64. Each of the IFU 60, the DOU 62, and the AOU 64 have a timing input coupled to the first timing signal line 40. The IFU 60 has a memory control output coupled to the memory control line 42, a data input coupled to the memory I/O line 46, and a bidirectional control port coupled to the external control line 48. The IFU 60 additionally has a first control output coupled to a first control input of the DOU 62 via a first control line 70, and a second control output coupled to a first control input of the AOU 64 via a second control line 72. The IFU 60 also has a third control output coupled to a second control input of the DOU 62 and a second control input of the AOU 64 via a third control line 74. The DOU 62 and the AOU 64 each have a bidirectional data port coupled to the memory I/O line 46. Finally, the AOU 64 has an address output that forms the DRPU's address output.

The DRPU 32 is preferably implemented using a reconfigurable or reprogrammable logic device, for example, an FPGA such as a Xilinx XC4013 (Xilinx, Inc., San Jose, CA) or an AT&T ORCA™ 1C07 (AT&T Microelectronics, Allentown, PA). Preferably, the reprogrammable logic device provides a plurality of: 1) selectively reprogrammable logic blocks, or Configurable Logic Blocks (CLBs); 2) selectively reprogrammable I/O Blocks (IOBs); 3) selectively reprogrammable interconnect structures; 4) data storage resources; 5) tri-state buffer resources; and 6) wired-logic function capabilities. Each CLB preferably includes selectively-reconfigurable circuitry for generating logic functions, storing data, and routing signals. Those skilled in the art will recognize that reconfigurable data storage circuitry may also be included in one or more Data Storage Blocks (DSBs) separate from the set of CLBs, depending upon the exact design of the reprogrammable logic device being used. Herein, the reconfigurable data storage circuitry within an FPGA is taken to be within the CLBs; that is, the presence of DSBs is not assumed. Those skilled in the art will readily recognize that one or more elements described herein that utilize CLB-based reconfigurable data storage circuitry could utilize DSB-based circuitry in the event that DSBs are present. Each IOB preferably includes selectively-reconfigurable circuitry for transferring data between CLBs and an FPGA output pin. A configuration data set defines a DRPU hardware configuration or organization by specifying functions performed within CLBs as well as interconnections: 1) within CLBs; 2) between CLBs; 3) within IOBs; 4) between IOBs; and 5) between CLBs and IOBs. Those skilled in the art will recognize that via a configuration data set, the number of bits in each of the memory control line 42, the address line 44, the memory I/O line 46, and the external control line 48 is reconfigurable. Preferably, configuration data sets are stored in one or more S-machine memories 34 within the system 10. Those skilled in the art will recognize that the DRPU 32 is

not limited to an FPGA-based implementation. For example, the DRPU 32 could be implemented as a RAM-based state machine that possibly includes one or more look-up tables. Alternatively, the DRPU 32 could be implemented using a Complex Programmable Logic Device (CPLD). However, those of ordinary skill in the art will realize that the some of the S-

5   machines 12 of the system 10 may have DRPUs 32 that are not reconfigurable.

In the preferred embodiment, the IFU 60, the DOU 62, and the AOU 64 are each dynamically reconfigurable. Thus, their internal hardware configuration can be selectively modified during program execution. The IFU 60 directs instruction fetch and decode operations, memory access operations, DRPU reconfiguration operations, and issues control signals to the

10   DOU 62 and the AOU 64 to facilitate instruction execution. The DOU 62 performs operations involving data computation, and the AOU 64 performs operations involving address computation. The internal structure and operation of each of the IFU 60, the DOU 62, and the AOU 64 will now be described in detail.

Referring now to Figure 5, a block diagram of a preferred embodiment of an Instruction

15   Fetch Unit 60 is shown. The IFU 60 comprises an Instruction State Sequencer (ISS) 100, an architecture description memory 101, memory access logic 102, reconfiguration logic 104, interrupt logic 106, a fetch control unit 108, an instruction buffer 110, a decode control unit 112, an instruction decoder 114, an opcode storage register set 116, a Register File (RF) address register set 118, a constants register set 120, and a process control register set 122. The ISS 100

20   has a first and a second control output that form the IFU's first and second control outputs, respectively, and a timing input that forms the IFU's timing input. The ISS 100 also has a fetch/decode control output coupled to a control input of the fetch control unit 108 and a control input of the decode control unit 112 via a fetch/decode control line 130. The ISS 100 additionally has a bidirectional control port coupled to a first bidirectional control port of each of

25   the memory access logic 102, the reconfiguration logic 104, and the interrupt logic 106 via a bidirectional control line 132. The ISS 100 also has an opcode input coupled to an output of the opcode storage register set 116 via an opcode line 142. Finally, the ISS 100 has a bidirectional data port coupled to a bidirectional data port of the process control register set 122 via a process data line 144.

30   Each of the memory access logic 102, the reconfiguration logic 104, and the interrupt logic 106 have a second bidirectional control port coupled to the external control line 48. The memory access logic 102, the reconfiguration logic 104, and the interrupt logic 106 additionally each have a data input coupled to a data output of the architecture description memory 101 via an implementation control line 131. The memory access logic 102 additionally has a control output

35   that forms the IFU's memory control output, and the interrupt logic 106 additionally has an output coupled to the process data line 144. The instruction buffer 110 has a data input that

forms the IFU's data input, a control input coupled to a control output of the fetch control unit 108 via a fetch control line 134, and an output coupled to an input of the instruction decoder 114 via an instruction line 136. The instruction decoder 114 has a control input coupled to a control output of the decode control unit 112 via a decode control line 138, and an output coupled via a decoded instruction line 140 to 1) an input of the opcode storage register set 116; 2) an input of the RF address register set 118; and 3) an input of the constants register set 120. The RF address register set 118 and the constants register set 120 each have an output that together form the IFU's third control output 74.

The architecture description memory 101 stores architecture specification signals that characterize the current DRPU configuration. Preferably, the architecture specification signals include 1) a reference to a default configuration data set; 2) a reference to a list of allowable configuration data sets; 3) a reference to a configuration data set corresponding to the currently-considered ISA, that is, a reference to the configuration data set that defines the current DRPU configuration; 4) an interconnect address list that identifies one or more interconnect I/O units 304 within the T-machine 14 associated with the S-machine 12 in which the IFU 60 resides, as will be described in detail below with reference to Figure 13; 5) a set of interrupt response signals that specify interrupt latency and interrupt precision information defining how the IFU 60 responds to interrupts; and 6) a memory access constant that defines an atomic memory address increment. In the preferred embodiment, each configuration data set implements the architecture description memory 101 as a set of CLBs configured as a Read-Only Memory (ROM). The architecture specification signals that define the contents of the architecture description memory 101 are preferably included in each configuration data set. Thus, because each configuration data set corresponds to a particular ISA, the contents of the architecture description memory 101 varies according to the ISA currently under consideration. For a given ISA, program access to the contents of the architecture description memory 101 is preferably facilitated by the inclusion of a memory read instruction in the ISA. This enables a program to retrieve information about the current DRPU configuration during program execution.

In the present invention, the reconfiguration logic 104 is a state machine that controls a sequence of reconfiguration operations that facilitate reconfiguration of the DRPU 32 according to a configuration data set. Preferably, the reconfiguration logic 104 initiates the reconfiguration operations upon receipt of a reconfiguration signal. As will be described in detail below, the reconfiguration signal is generated by the interrupt logic 106 in response to a reconfiguration interrupt received on the external control line 48, or by the ISS 100 in response to a reconfiguration directive embedded within a program. The reconfiguration operations provide for an initial DRPU configuration following a power-on/reset condition using the default configuration data set referenced by the architecture description memory 101. The

reconfiguration operations also provide for selective DRPU reconfiguration after the initial DRPU configuration has been established. Upon completion of the reconfiguration operations, the reconfiguration logic 104 issues a completion signal. In the preferred embodiment, the reconfiguration logic 104 is non-reconfigurable logic that controls the loading of configuration

5   data sets into the reprogrammable logic device itself, and thus the sequence of reconfiguration operations is defined by the reprogrammable logic device manufacturer. The reconfiguration operations will therefore be known to those skilled in the art.

Each DRPU configuration is preferably given by a configuration data set that defines a particular hardware organization dedicated to the implementation of a corresponding ISA. In the

10   preferred embodiment, the IFU 60 includes each of the elements indicated above, regardless of DRPU configuration. At a basic level, the functionality provided by each element within the IFU 60 is independent of the currently-considered ISA. However, in the preferred embodiment, the detailed structure and functionality of one or more elements of the IFU 60 may vary based upon the nature of the ISA for which it has been configured. In the preferred embodiment, the

15   structure and functionality of the architecture description memory 101 and the reconfiguration logic 104 preferably remain constant from one DRPU configuration to another. The structure and functionality of the other elements of the IFU 60 and the manner in which they vary according to ISA type will now be described in detail.

The process control register set 122 stores signals and data used by the ISS 100 during

20   instruction execution. In the preferred embodiment, the process control register set 122 comprises a register for storing a process control word, a register for storing an interrupt vector, and a register for storing a reference to a configuration data set. The process control word preferably includes a plurality of condition flags that can be selectively set and reset based upon conditions that occur during instruction execution. The process control word additionally

25   includes a plurality of transition control signals that define one or more manners in which interrupts can be serviced, as will be described in detail below. In the preferred embodiment, the process control register set 122 is implemented as a set of CLBs configured for data storage and gating logic.

The ISS 100 is preferably a state machine that controls the operation of the fetch control

30   unit 108, the decode control unit 112, the DOU 62 and the AOU 64, and issues memory read and memory write signals to the memory access logic 102 to facilitate instruction execution. Referring now to Figure 6, a state diagram showing a preferred set of states supported by the ISS 100 is shown. Following a power-on or reset condition, or immediately after reconfiguration has occurred, the ISS 100 begins operation in state P. In response to the completion signal issued by

35   the reconfiguration logic 104, the ISS 100 proceeds to state S, in which the ISS initializes or restores program state information in the event that a power-on/reset condition or a

reconfiguration has occurred, respectively. The ISS 100 next advances to state F, in which instruction fetch operations are performed. In the instruction fetch operations, the ISS 100 issues a memory read signal to the memory access logic 102, issues a fetch signal to the fetch control unit 108, and issues an increment signal to the AOU 64 to increment a Next Instruction Program

5 Address Register (NIPAR) 232, as will be described in detail below with reference to Figures 11A and 11B. After state F, the ISS 100 advances to state D to initiate instruction decoding operations. In state D, the ISS 100 issues a decode signal to the decode control unit 112. While in state D, the ISS 100 additionally retrieves an opcode corresponding to a decoded instruction from the opcode storage register set 116. Based upon the retrieved opcode, the ISS 100 proceeds

10 to state E or to state M to perform instruction execution operations. The ISS 100 advances to state E in the event that the instruction can be executed in a single clock cycle; otherwise, the ISS 100 advances to state M for multicycle instruction execution. In the instruction execution operations, the ISS 100 generates DOU control signals, AOU control signals, and/or signals directed to the memory access logic 102 to facilitate the execution of the instruction

15 corresponding to the retrieved opcode. Following either of states E or M, the ISS 100 advances to state W. In state W, the ISS 100 generates DOU control signals, AOU control signals, and/or memory write signals to facilitate storage of an instruction execution result. State W is therefore referred to as a write-back state. Those skilled in the art will recognize that states F, D, E or M, and W comprise a complete instruction execution cycle. After state W, the ISS 100 advances to

20 state Y in the event that suspension of instruction execution is required. State Y corresponds to an idle state, which may be required, for example, in the event that a T-machine 14 requires access to the S-machine's memory 34. Following state Y, or after state W in the event that instruction execution is to continue, the ISS 100 returns to state F to resume another instruction execution cycle.

25 As shown in Figure 6, the state diagram also includes state I, which is defined to be an interrupt service state. In the present invention, the ISS 100 receives interrupt notification signals from the interrupt logic 106. As will be described in detail below with reference to Figure 7, the interrupt logic 106 generates transition control signals, and stores the transition control signals in the process control word within the process control register set 122. The

30 transition control signals preferably indicate which of the states F, D, E, M, W, and Y are interruptable, a level of interrupt precision required in each interruptable state, and for each interruptable state a next state at which instruction execution is to continue following state I. If the ISS 100 receives an interrupt notification signal while in a given state, the ISS 100 advances to state I if the transition control signals indicate that the current state is interruptable.

35 Otherwise, the ISS 100 advances as if no interrupt signal has been received, until reaching an interruptable state.

Once the ISS 100 has advanced to state I, the ISS 100 preferably accesses the process control register set 122 to set an interrupt masking flag and retrieve an interrupt vector. After retrieving the interrupt vector, the ISS 100 preferably services the current interrupt via a conventional subroutine jump to an interrupt handler as specified by the interrupt vector.

5   In the present invention, reconfiguration of the DRPU 32 is initiated in response to 1) a reconfiguration interrupt asserted upon the external control line 48; or 2) the execution of a reconfiguration directive within a sequence of program instructions. In the preferred embodiment, both the reconfiguration interrupt and the execution of a reconfiguration directive result in a subroutine jump to a reconfiguration handler. Preferably, the reconfiguration handler

10   saves program state information, and issues a configuration data set address and the reconfiguration signal to the reconfiguration logic 104.

In the event that the current interrupt is not a reconfiguration interrupt, the ISS 100 advances to a next state as indicated by the transition control signals once the interrupt has been serviced, thereby resuming, completing, or initiating an instruction execution cycle.

15   In the preferred embodiment, the set of states supported by the ISS 100 varies according to the nature of the ISA for which the DRPU 32 is configured. Thus, state M would not be present for an ISA in which one or more instructions can be executed in a single clock cycle, as would be the case with a typical inner-loop ISA. As depicted, the state diagram of Figure 6 preferably defines the states supported by the ISS 100 for implementing a general-purpose outer-

20   loop ISA. For the implementation of an inner-loop ISA, the ISS 100 preferably supports multiple sets of states F, D, E, and W in parallel, thereby facilitating pipelined control of instruction execution in a manner that will be readily understood by those skilled in the art. In the preferred embodiment, the ISS 100 is implemented as a CLB-based state machine that supports the states or a subset of the states described above, in accordance with the currently-

25   considered ISA.

The interrupt logic 106 preferably comprises a state machine that generates transition control signals, and performs interrupt notification operations in response to an interrupt signal received via the external control line 48. Referring now to Figure 7, a state diagram showing a preferred set of states supported by the interrupt logic 106 is shown. The interrupt logic 106

30   begins operation in state P. State P corresponds to a power-on, reset, or reconfiguration condition. In response to the completion signal issued by the reconfiguration logic 104, the interrupt logic 106 advances to state A and retrieves the interrupt response signals from the architecture description memory 101. The interrupt logic 106 then generates the transition control signals from the interrupt response signals, and stores the transition control signals in the

35   process control register set 122. In the preferred embodiment, the interrupt logic 106 includes a CLB-based Programmable Logic Array (PLA) for receiving the interrupt response signals and

generating the transition control signals. Following state A, the interrupt logic 106 advances to state B to wait for an interrupt signal. Upon receipt of an interrupt signal, the interrupt logic 106 advances to state C in the event that the interrupt masking flag within the process control register set 122 is reset. Once in state C, the interrupt logic 106 determines the origin of the interrupt, an

5   interrupt priority, and an interrupt handler address. In the event that the interrupt signal is a reconfiguration interrupt, the interrupt logic 106 advances to state R and stores a configuration data set address in the process control register set 122. After state R, or following state C in the event that the interrupt signal is not a reconfiguration interrupt, the interrupt logic 106 advances to state N and stores the interrupt handler address in the process control register set 122. The

10   interrupt logic 106 next advances to state X, and issues an interrupt notification signal to the ISS 100. Following state X, the interrupt logic 122 returns to state B to wait for a next interrupt signal.

In the preferred embodiment, the level of interrupt latency as specified by the interrupt response signals, and hence the transition control signals, varies according to the current ISA for

15   which the DRPU 32 has been configured. For example, an ISA dedicated to high-performance real-time motion control requires rapid and predictable interrupt response capabilities. The configuration data set corresponding to such an ISA therefore preferably includes interrupt response signals that indicate low-latency interruption is required. The corresponding transition control signals in turn preferably identify multiple ISS states as interruptable, thereby allowing

20   an interrupt to suspend an instruction execution cycle prior to the instruction execution cycle's completion. In contrast to an ISA dedicated to real-time motion control, an ISA dedicated to image convolution operations requires interrupt response capabilities that ensure that the number of convolution operations performed per unit time is maximized. The configuration data set corresponding to the image convolution ISA preferably includes interrupt response signals that

25   specify high-latency interruption is required. The corresponding transition control signals preferably identify state W as being interruptable. In the event that the ISS 100 supports multiple sets of states F, D, E, and W in parallel when configured to implement the image convolution ISA, the transition control signals preferably identify each state W as being interruptable, and further specify that interrupt servicing is to be delayed until each of the parallel instruction

30   execution cycles have completed their state W operations. This ensures that an entire group of instructions will be executed before an interrupt is serviced, thereby maintaining reasonable pipelined execution performance levels.

In a manner analogous to the level of interrupt latency, the level of interrupt precision as specified by the interrupt response signals also varies according to the ISA for which the DRPU

35   32 is configured. For example, in the event that state M is defined to be an interruptable state for an outer-loop ISA that supports interruptable multicycle operations, the interrupt response

signals preferably specify that precise interrupts are required. The transition control signals thus specify that interrupts received in state M are treated as precise interrupts to ensure that multicycle operations can be successfully restarted. As another example, for an ISA which supports nonfaultable pipelined arithmetic operations, the interrupt response signals preferably

5   specify that imprecise interrupts are required. The transition control signals then specify that interrupts received in state W are treated as imprecise interrupts.

For any given ISA, the interrupt response signals are defined, or programmed, by a portion of the ISA's corresponding configuration data set. Via the programmable interrupt response signals and the generation of corresponding transition control signals, the present

10   invention facilitates the implementation of an optimum interruption scheme on an ISA-by-ISA basis. Those skilled in the art will recognize that the vast majority of prior art computer architectures do not provide for the flexible specification of interruption capabilities, namely, programmable state transition enabling, programmable interrupt latency, and programmable interrupt precision. In the preferred embodiment, the interrupt logic 106 is implemented as a

15   CLB-based state machine that supports the states described above.

The fetch control unit 108 directs the loading of instructions into the instruction buffer 110 in response to the fetch signal issued by the ISS 100. In the preferred embodiment, the fetch control unit 108 is implemented as a conventional one-hot encoded state machine using flip-flops within a set of CLBs. Those skilled in the art will recognize that in an alternate embodiment, the

20   fetch control unit 108 could be configured as a conventional encoded state machine or as a ROM-based state machine. The instruction buffer 110 provides temporary storage for instructions loaded from the memory 34. For the implementation of an outer-loop ISA, the instruction buffer 110 is preferably implemented as a conventional RAM-based First In, First Out (FIFO) buffer using a plurality of CLBs. For the implementation of an inner-loop ISA, the

25   instruction buffer 110 is preferably implemented as a set of flip-flop registers using a plurality of flip-flops within a set of IOBs or a plurality of flip-flops within both IOBs and CLBs.

The decode control unit 112 directs the transfer of instructions from the instruction buffer 110 into the instruction decoder 114 in response to the decode signal issued by the ISS 100. For an inner-loop ISA, the decode control unit 112 is preferably implemented as a ROM-based state

30   machine comprising a CLB-based ROM coupled to a CLB-based register. For an outer-loop ISA, the decode control unit 112 is preferably implemented as a CLB-based encoded state machine. For each instruction received as input, the instruction decoder 114 outputs a corresponding opcode, a register file address, and optionally one or more constants in a conventional manner. For an inner-loop ISA, the instruction decoder 114 is preferably

35   configured to decode a group of instructions received as input. In the preferred embodiment, the

instruction decoder 114 is implemented as a CLB-based decoder configured to decode each of the instructions included in the ISA currently under consideration.

The opcode storage register set 116 provides temporary storage for each opcode output by the instruction decoder 144, and outputs each opcode to the ISS 100. When an outer-loop ISA is

5 implemented in the DRPU 32, the opcode storage register set 116 is preferably implemented using an optimum number of flip-flop register banks. The flip-flop register banks receive signals from the instruction decoder 114 that represent class or group codes derived from opcode literal bitfields from instructions previously queued through the instruction buffer 110. The flip-flop register banks store the aforementioned class or group codes according to a decoding scheme that

10 preferably minimizes ISS complexity. In the case of an inner-loop ISA, the opcode storage register set 116 preferably stores opcode indication signals that are more directly derived from opcode literal bitfields output by the instruction decoder 114. Inner-loop ISAs necessarily have smaller opcode literal bitfields, thereby minimizing the implementation requirements for buffering, decoding, and opcode indication for instruction sequencing by the instruction buffer

15 110, the instruction decoder 114, and the opcode storage register set 116, respectively. In summary, for outer-loop ISAs, the opcode storage register set 116 is preferably implemented as a small federation of flip-flop register banks characterized by a bitwidth equal to or a fraction of the opcode literal size. For inner-loop ISAs, the opcode storage register set 116 is preferably a smaller and more unified flip-flop register bank than for outer-loop ISAs. The reduced flip-flop

20 register bank size in the inner-loop case reflects the minimal instruction count characteristic of inner-loop ISAs relative to outer-loop ISAs.

The RF address register set 118 and the constants register set 120 provide temporary storage for each register file address and each constant output by the instruction decoder 114, respectively. In the preferred embodiment, the opcode storage register set 116, the RF address

25 register set 118, and the constants register set 120 are each implemented as a set of CLBs configured for data storage.

The memory access logic 102 is memory control circuitry that directs and synchronizes the transfer of data between the memory 34, the DOU 62, and the AOU 64 according to the atomic memory address size specified in the architecture description memory 122. The memory

30 access logic 102 additionally directs and synchronizes the transfer of data and commands between the S-machine 12 and a given T-machine 14. In the preferred embodiment, the memory access logic 102 supports burst-mode memory accesses, and is preferably implemented as a conventional RAM controller using CLBs. Those skilled in the art will recognize that during reconfiguration, the input and output pins of the reconfigurable logic device will be three-stated,

35 allowing resistive terminations to define unasserted logic levels, and hence will not perturb the

memory 34. In an alternate embodiment, the memory access logic 102 could be implemented external to the DRPU 32.

Referring now to Figure 8, a block diagram of a preferred embodiment of the Data Operate Unit 62 is shown. The DOU 62 performs operations upon data according to DOU
5   control signals, RF addresses, and constants received from the ISS 100. The DOU 62 comprises a DOU cross-bar switch 150, store/align logic 152, and data operate logic 154. Each of the DOU cross-bar switch 150, the store/align logic 152, and the data operate logic 154 have a control input coupled to the first control output of the IFU 60 via the first control line 70. The DOU cross-bar switch 150 has a bidirectional data port that forms the DOU's bidirectional data port; a
10 constants input coupled to the third control line 74; a first data feedback input coupled to a data output of the data operate logic 154 via a first data line 160; a second data feedback input coupled to a data output of the store/align logic 152 via a second data line 164; and a data output coupled to a data input of the store/align logic 152 via a third data line 162. In addition to its data output, the store/align logic 154 has an address input coupled to the third control line 74.
15 The data operate logic 154 additionally has a data input coupled to the store/align logic's output via the second data line 164.

The data operate logic 154 performs arithmetic, shifting, and/or logical operations upon data received at its data input in response to the DOU control signals received at its control input. The store/align logic 152 comprises data storage elements that provide temporary storage for
20 operands, constants, and partial results associated with data computations, under the direction of RF addresses and DOU control signals received at its address input and control input, respectively. The DOU cross-bar switch 150 is preferably a conventional cross-bar switch network that facilitates the loading of data from the memory 34, the transfer of results output by the data operate logic 154 to the store/align logic 152 or the memory 34, and the loading of
25 constants output by the IFU 60 into the store/align logic 152 in accordance with the DOU control signals received at its control input. In the preferred embodiment, the detailed structure of the data operate logic 154 is dependent upon the types of operations supported by the ISA currently under consideration. That is, the data operate logic 154 comprises circuitry for performing the arithmetic and/or logical operations specified by the data-operate instructions within the
30 currently-considered ISA. Similarly, the detailed structure of the store/align logic 152 and the DOU cross-bar switch 150 is dependent upon the ISA currently under consideration. The detailed structure of the data operate logic 154, the store/align logic 152, and the DOU cross-bar switch 150 according to ISA type is described hereafter with reference to Figures 9A and 9B.

For an outer-loop ISA, the DOU 62 is preferably configured to perform serial operations
35 upon data. Referring now to Figure 9A, a block diagram of a first exemplary embodiment of the DOU 61 configured for the implementation of a general-purpose outer-loop ISA is shown. A

general-purpose outer-loop ISA requires hardware configured for performing mathematical operations such as multiplication, addition, and subtraction; Boolean operations such as AND, OR, and NOT; shifting operations; and rotating operations. Thus, for the implementation of a general-purpose outer-loop ISA, the data operate logic 154 preferably comprises a conventional

5   Arithmetic-Logic Unit (ALU)/shifter 184 having a first input, a second input, a control input, and an output. The Store/Align logic 152 preferably comprises a first RAM 180 and a second RAM 182, each of which has a data input, a data output, an address-select input, and an enable input. The DOU cross-bar switch 150 preferably comprises a conventional cross-bar switch network having both bidirectional and unidirectional crossbar couplings, and having the inputs and

10   outputs previously described with reference to Figure 8. Those skilled in the art will recognize that an efficient implementation of the DOU cross-bar switch 150 for an outer-loop ISA may include multiplexors, tri-state buffers, CLB-based logic, direct wiring, or subsets of the aforementioned elements joined in any combination by virtue of reconfigurable coupling means. For an outer-loop ISA, the DOU cross-bar switch 150 is implemented to expedite serial data

15   movement in a minimum possible time, while also providing a maximum number of unique data movement cross-bar couplings to support generalized outer-loop instruction types.

The data input of the first RAM 180 is coupled to the data output of the DOU cross-bar switch 150, as is the data input of the second RAM 182, via the third data line 162. The address-select inputs of the first RAM 180 and the second RAM 182 are coupled to receive register file

20   addresses from the IFU 60 via the third control line 74. Similarly, the enable inputs of the first and second RAM 180, 182 are coupled to receive DOU control signals via the first control line 70. The data outputs of the first and second RAM 180, 182 are coupled to the first input and the second input of the ALU/shifter 184, respectively, and are also coupled to the second data feedback input of the DOU cross-bar switch 150. The control input of the ALU/shifter 184 is

25   coupled to receive DOU control signals via the first control line 70, and the output of the ALU/shifter 184 is coupled to the first data feedback input of the DOU cross-bar switch 150. The couplings to the remaining inputs and outputs of the DOU cross-bar switch 150 are identical to those given in the description above with reference to Figure 8.

To facilitate the execution of a data-operate instruction, the IFU 60 issues DOU control

30   signals, RF addresses, and constants to the DOU 61 during either of ISS states E or M. The first and second RAM 180, 182 provide a first and second register file for temporary data storage, respectively. Individual addresses within the first and second RAM 180, 182 are selected according to the RF addresses received at each RAM's respective address-select input. Similarly, loading of the first and second RAM 180, 182 is controlled by the DOU control signals each

35   respective RAM 180, 182 receives at its write-enable input. In the preferred embodiment, at least one RAM 180, 182 includes a pass-through capability to facilitate the transfer of data from

the DOU cross-bar switch 150 directly into the ALU/shifter 184. The ALU/shifter 184 performs arithmetic, logical, or shifting operations upon a first operand received from the first RAM 180 and/or a second operand received from the second RAM 182, under the direction of the DOU control signals received at its control input. The DOU cross-bar switch 150 selectively routes: 1)

5   data between the memory 34 and the first and second RAM 180, 182; 2) results from the ALU/shifter 184 to the first and second RAM 180, 182 or the memory 34; 3) data stored in the first or second RAM 180, 182 to the memory 34; and 4) constants from the IFU 60 to the first and second RAM 180, 182. As previously described, in the event that either the first or second RAM 180, 182 includes a pass-through capability, the DOU cross-bar switch 150 also selectively

10   routes data from the memory 34 or the ALU/shifter's output directly back into the ALU/shifter 184. The DOU cross-bar switch 150 performs a particular routing operation according to the DOU control signals received at its control input. In the preferred embodiment, the ALU/shifter 184 is implemented using logic function generators within a set of CLBs and circuitry dedicated to mathematical operations within the reconfigurable logic device. The first and second RAM

15   180, 182 are each preferably implemented using the data storage circuitry present within a set of CLBs, and the DOU cross-bar switch 150 is preferably implemented in the manner previously described.

Referring now to Figure 9B, a block diagram of a second exemplary embodiment of the DOU 63 configured for the implementation of an inner-loop ISA is shown. In general, an inner-

20   loop ISA supports relatively few, specialized operations, and is preferably used to perform a common set of operations upon potentially large data sets. Optimum computational performance for an inner-loop ISA is therefore produced by hardware configured to perform operations in parallel. Thus, in the second exemplary embodiment of the DOU 63, the data operate logic 154, the store/align logic 152, and the DOU cross-bar switch 150 are configured to perform pipelined

25   computations. The data operate logic 154 comprises a pipelined functional unit 194 having a plurality of inputs, a control input, and an output. The store/align logic 152 comprises: 1) a set of conventional flip-flop arrays 192, each flip-flop array 192 having a data input, a data output, and a control input; and 2) a data selector 190 having a control input, a data input, and a number of data outputs corresponding to the number of flip-flop arrays 192 present. The DOU cross-bar

30   switch 150 comprises a conventional cross-bar switch network having duplex unidirectional crossbar couplings. In the second exemplary embodiment of the DOU 63, the DOU cross-bar switch 150 preferably includes the inputs and outputs previously described with reference to Figure 8, with the exception of the second data feedback input. In a manner analogous to the outer-loop ISA case, an efficient implementation of the DOU cross-bar switch 150 for an inner-

35   loop ISA may include multiplexors, tri-state buffers, CLB-based logic, direct wiring, or a subset of the aforementioned elements coupled in a reconfigurable manner. For an inner-loop ISA, the

DOU cross-bar switch 150 is preferably implemented to maximize parallel data movement in a minimum amount of time, while also providing a minimum number of unique data movement cross-bar couplings to support heavily pipelined inner-loop ISA instructions.

The data input of the data selector 190 is coupled to the data output of the DOU cross-bar
5   switch 150 via the first data line 162. The control input of the data selector 190 is coupled to receive RF addresses via the third control line 74, and each output of the data selector 190 is coupled to a corresponding flip-flop array data input. The control input of each flip-flop array 192 is coupled to receive DOU control signals via the first control line 70, and each flip-flop array data output is coupled to an input of the functional unit 194. The control input of the
10   functional unit 194 is coupled to receive DOU control signals via the first control line 70, and the output of the functional unit 194 is coupled to the first data feedback input of the DOU cross-bar switch 150. The couplings of the remaining inputs and outputs of the DOU cross-bar switch 150 are identical to those previously described with reference to Figure 8.

In operation, the functional unit 194 performs pipelined operations upon data received at
15   its data inputs in accordance with the DOU control signals received at its control input. Those skilled in the art will recognize that the functional unit 194 may be a multiply-accumulate unit, a threshold determination unit, an image rotation unit, an edge enhancement unit, or any type of functional unit suitable for performing pipelined operations upon partitioned data. The data selector 190 routes data from the output of the DOU cross-bar switch 150 into a given flip-flop
20   array 192 according to the RF addresses received at its control input. Each flip-flop array 192 preferably includes a set of sequentially-coupled data latches for spatially and temporally aligning data relative to the data contents of another flip-flop array 192, under the direction of the control signals received at its control input. The DOU cross-bar switch 150 selectively routes: 1) data from the memory 34 to the data selector 190; 2) results from the multiply/accumulate unit
25   194 to the data selector 190 or the memory 34; and 3) constants from the IFU 60 to the data selector 190. Those skilled in the art will recognize that an inner-loop ISA may have a set of "built-in" constants. In the implementation of such an inner-loop ISA, the store/align logic 154 preferably includes a CLB-based ROM containing the built-in constants, thereby eliminating the need to route constants from the IFU 60 into the store/align logic 152 via the DOU cross-bar
30   switch 150. In the preferred embodiment, the functional unit 194 is preferably implemented using logic function generators and circuitry dedicated to mathematical operations within a set of CLBs. Each flip-flop array 192 is preferably implemented using flip-flops within a set of CLBs, and the data selector 190 is preferably implemented using logic function generators and data selection circuitry within a set of CLBs. Finally, the DOU cross-bar switch 150 is preferably
35   implemented in the manner previously described for an inner-loop ISA.

Referring now to Figure 10, a block diagram of a preferred embodiment of the Address Operate Unit 64 is shown. The AOU 64 performs operations upon addresses according to AOU control signals, RF addresses, and constants received from the IFU 60. The AOU 64 comprises a AOU cross-bar switch 200, store/count logic 202, address operate logic 204, and an address

5   multiplexor 206. Each of the AOU cross-bar switch 200, the store/count logic 202, the address operate logic 204, and the address multiplexor 206 has a control input coupled to the second control output of the IFU 60 via the second control line 72. The AOU cross-bar switch 200 has a bidirectional data port that forms the AOU's bidirectional data port; an address feedback input coupled to an address output of the address operate logic 204 via a first address line 210; a

10   constants input coupled to the third control line 74; and an address output coupled to an address input of the store/count logic 202 via a second address line 212. In addition to its address input and control input, the store/count logic 202 has an RF address input coupled to the third control line 74, and an address output coupled to an address input of the address operate logic 204 via a third address line 214. The address multiplexor 206 has a first input coupled to the first address

15   line 210, a second input coupled to the third address line 214, and an output that forms the address output of the AOU 64.

The address operate logic 204 performs arithmetic operations upon addresses received at its address input under the direction of AOU control signals received at its control input. The store/count logic 202 provides temporary storage of addresses and address computation results.

20   The AOU cross-bar switch 200 facilitates the loading of addresses from the memory 34, the transfer of results output by the address operate logic 204 to the store/count logic 202 or the memory 34, and the loading of constants output by the IFU 60 into the store/count logic 202 in accordance with the AOU control signals received at its control input. The address multiplexor 206 selectively outputs an address received from the store/count logic 202 or the address operate

25   logic 200 to the address output of the AOU 64 under the direction of the AOU control signals received at its control input. In the preferred embodiment, the detailed structure of the AOU cross-bar switch 200, the store/align logic 202, and the address operate unit 204 is dependent upon the type of ISA currently under consideration, as is described hereafter with reference to Figures 11A and 11B.

30   Referring now to Figure 11A, a block diagram of a first exemplary embodiment of the AOU 65 configured for the implementation of a general-purpose outer-loop ISA is shown. A general-purpose outer-loop ISA requires hardware for performing operations such as addition, subtraction, increment, and decrement upon the contents of a program counter and addresses stored in the store/count logic 202. In the first exemplary embodiment of the AOU 65, the

35   address operate logic 204 preferably comprises a Next Instruction Program Address Register (NIPAR) 232 having an input, an output, and a control input; an arithmetic unit 234 having a first

input, a second input, a third input, a control input, and an output; and a multiplexor 230 having a first input, a second input, a control input, and an output. The store/count logic 202 preferably comprises a third RAM 220 and a fourth RAM 222, each of which has an input, an output, an address-select input, and an enable input. The address multiplexor 206 preferably comprises a

5 multiplexor having a first input, a second input, a third input, a control input, and an output. The AOU cross-bar switch 200 preferably comprises a conventional cross-bar switch network having duplex unidirectional crossbar couplings, and having the inputs and outputs previously described with reference to Figure 10. An efficient implementation of the AOU cross-bar switch 200 may include multiplexors, tri-state buffers, CLB-based logic, direct wiring, or any subset of such

10 elements joined by reconfigurable couplings. For an outer-loop ISA, the AOU cross-bar switch 200 is preferably implemented to maximize serial address movement in a minimum amount of time, while also providing a maximum number of unique address movement cross-bar couplings to support generalized outer-loop ISA address operate instructions.

The input of the third RAM 220 and the input of the fourth RAM 222 are each coupled to

15 the output of the AOU cross-bar switch 200 via the second address line 212. The address-select inputs of the third and fourth RAM 220, 222 are coupled to receive RF addresses from the IFU 60 via the third control line 74, and the enable inputs of the first and second RAM 220, 222 are coupled to receive AOU control signals via the second control line 72. The output of the third RAM 220 is coupled to the first input of the multiplexor 230, the first input of the arithmetic unit

20 234, and the first input of the address multiplexor 206. Similarly, the output of the fourth RAM 222 is coupled to the second input of the multiplexor 230, the second input of the arithmetic unit 234, and the second input of the address multiplexor 206. The control inputs of the multiplexor 230, the NIPAR 232, and the arithmetic unit 234 are each coupled to the second control line 72. The output of the arithmetic unit 234 forms the output of the address operate logic 204, and is

25 therefore coupled to the address feedback input of the AOU cross-bar switch 200 and the third input of the address multiplexor 206. The couplings to the remaining inputs and outputs of the AOU cross-bar switch 200 and the address multiplexor 206 are identical to those previously described with reference to Figure 10.

To facilitate the execution of an address-operate instruction, the IFU 60 issues AOU

30 control signals, RF addresses, and constants to the AOU 64 during either of ISS states E or M. The third and fourth RAM 220, 222 provide a first and a second register file for temporary address storage, respectively. Individual storage locations within the third and fourth RAM 220, 222 are selected according to the RF addresses received at each RAM's respectively address-select input. The loading of the third and fourth RAM 220, 222 is controlled by the AOU control

35 signals each respective RAM 220, 222 receives at its write-enable input. The multiplexor 230 selectively routes addresses output by the third and fourth RAM 220, 222 to the NIPAR 232

under the direction of the AOU control signals received at its control input. The NIPAR 232 loads an address received from the output of the multiplexor 230 and increments its contents in response to the AOU control signals received at its control input. In the preferred embodiment, the NIPAR 232 stores the address of the next program instruction to be executed. The arithmetic

5   unit 234 performs arithmetic operations including addition, subtraction, increment, and decrement upon addresses received from the third and fourth RAM 220, 222 and/or upon the contents of the NIPAR 232. The AOU cross-bar switch 200 selectively routes: 1) addresses from the memory 34 to the third and fourth RAM 220, 222; and 2) results of address computations output by the arithmetic unit 234 to the memory 34 or the third and fourth RAM 220, 222. The

10   AOU cross-bar switch 200 performs a particular routing operation according to the AOU control signals received at its control input. The address multiplexor 206 selectively routes addresses output by the third RAM 220, addresses output by the fourth RAM 222, or the results of address computations output by the arithmetic unit 234 to the AOU's address output under the direction of the AOU control signals received at its control input.

15   In the preferred embodiment, the third and fourth RAM 220, 222 are each implemented using the data storage circuitry present within a set of CLBs. The multiplexor 230 and the address multiplexor 206 are each preferably implemented using data selection circuitry present within a set of CLBs, and the NIPAR 232 is preferably implemented using data storage circuitry present within a set of CLBs. The arithmetic unit 234 is preferably implemented using logic

20   function generators and circuitry dedicated to mathematical operations within a set of CLBs. Finally, the AOU cross-bar switch 200 is preferably implemented in the manner previously described.

Referring now to Figure 11B, a block diagram of a second exemplary embodiment of the AOU 66 configured for the implementation of an inner-loop ISA is shown. Preferably, an inner-

25   loop ISA requires hardware for performing a very limited set of address operations, and hardware for maintaining at least one source address pointer and a corresponding number of destination address pointers. Types of inner-loop processing for which a very limited number of address operations or even a single address operation are required include block, raster, or serpentine operations upon image data; bit reversal operations; operations upon circular buffer data; and

30   variable length data parsing operations. Herein, a single address operation is considered, namely, an increment operation. Those skilled in the art will recognize that hardware that performs increment operations may also be inherently capable of performing decrement operations, thereby providing an additional address operation capability. In the second exemplary embodiment of the AOU 66, the store/count logic 202 comprises at least one source address

35   register 252 having an input, an output, and a control input; at least one destination address register 254 having an input, an output, and a control input; and a data selector 250 having an

input, a control input, and a number of outputs equal to the total number of source and destination address registers 252, 254 present. Herein, a single source address register 252 and a single destination address register 254 are considered, and hence the data selector 250 has a first output and a second output. The address operate logic 204 comprises a NIPAR 232 having an

5   input, an output, and a control output; and a multiplexor 260 having a number of inputs equal to the number of data selector outputs, a control input, and an output. Herein, the multiplexor 260 has a first input and a second input. The address multiplexor 206 preferably comprises a multiplexor having a number of inputs one greater than the number of data selector outputs, a control input, and an output. Thus, herein the address multiplexor 206 has a first input, a second

10   input, and a third input. The AOU cross-bar switch 200 preferably comprises a conventional cross-bar switch network having bidirectional and unidirectional crossbar couplings, and having the inputs and outputs previously described with reference to Figure 10. An efficient implementation of the AOU cross-bar switch 200 may include multiplexors, tri-state buffers, CLB-based logic, direct wiring, or any subset of such elements joined by reconfigurable

15   couplings. For an inner-loop ISA, the AOU cross-bar switch 200 is preferably implemented to maximize parallel address movement in a minimum possible time, while also providing a minimum number of unique address movement cross-bar couplings to support inner-loop address operations.

The input of the data selector 250 is coupled to the output of the AOU cross-bar switch

20   200. The first and second outputs of the data selector 250 are coupled to the input of the source address register 252 and the input of the destination address register 254, respectively. The control inputs of the source address register 252 and the destination address register 254 are coupled to receive AOU control signals via the second control line 72. The output of the source address register 252 is coupled to the first input of the multiplexor 260 and the first input of the

25   address multiplexor 206. Similarly, the output of the destination register 254 is coupled to the second input of the multiplexor 254 and the second input of the address multiplexor 206. The input of the NIPAR 232 is coupled to the output of the multiplexor 260, the control input of the NIPAR 232 is coupled to receive AOU control signals via the second control line 72, and the output of the NIPAR 232 is coupled to both the address feedback input of the AOU cross-bar

30   switch 200 and the third input of the address multiplexor 206. The couplings to the remaining inputs and outputs of the AOU cross-bar switch 200 are identical to those previously described above with reference to Figure 10.

In operation, the data selector 250 routes addresses received from the AOU cross-bar switch to the source address register 252 or the destination address register 254 according to the

35   RF addresses received at its control input. The source address register 252 loads an address present at its input in response to the AOU control signals present at its control input. The

destination address 254 register loads an address present at its input in an analogous manner. The multiplexor 260 routes an address received from the source address register 252 or the destination address register 254 to the input of the NIPAR 232 according to the AOU control signals received at its control input. The NIPAR 232 loads an address present at its input,

5   increments its contents, or decrements its contents in response to the AOU control signals received at its control input. The AOU cross-bar switch 200 selectively routes: 1) addresses from the memory 34 to the data selector 250; and 2) the contents of the NIPAR 232 to the memory 34 or the data selector 250. The AOU cross-bar switch 200 performs a particular routing operation according to the AOU control signals received at its control input. The address multiplexor 206

10   selectively routes the contents of the source address register 252, the destination address register 254, or the NIPAR 232 to the AOU's address output under the direction of the AOU control signals received at its control input.

In the preferred embodiment, the source address register 252 and the destination address register 254 are each implemented using the data storage circuitry present within a set of CLBs.

15   The NIPAR 232 is preferably implemented using increment/decrement logic and flip-flops within a set of CLBs. The data selector 250, the multiplexor 230, and the address multiplexor 206 are each preferably implemented using data selection circuitry present within a set of CLBs. Finally, the AOU cross-bar switch 200 is preferably implemented in the manner previously described for an inner-loop ISA. Those skilled in the art will recognize that in certain

20   applications, it may be advantageous to utilize an ISA that relies upon an inner-loop AOU configuration with an outer-loop DOU configuration, or vice-versa. For example, an associative string search ISA would beneficially utilize an inner-loop DOU configuration with an outer-loop AOU configuration. As another example, an ISA for performing histogram operations would beneficially utilize an outer-loop DOU configuration with an inner-loop AOU configuration.

25   Finite reconfigurable hardware resources must be allocated between each element of the DRPU 32. Because the reconfigurable hardware resources are limited in number, the manner in which they are allocated to the IFU 60, for example, affects the maximum computational performance level achievable by the DOU 62 and the AOU 64. The manner in which the reconfigurable hardware resources are allocated between the IFU 60, the DOU 62, and the AOU

30   64 varies according to the type of ISA to be implemented at any given moment. As ISA complexity increases, more reconfigurable hardware resources must be allocated to the IFU 60 to facilitate increasingly complex decoding and control operations, leaving fewer reconfigurable hardware resources available between the DOU 62 and the AOU 64. Thus, the maximum computational performance achievable from the DOU 62 and the AOU 64 decreases with ISA

35   complexity. In general, an outer-loop ISA will have many more instructions than an inner-loop ISA, and therefore its implementation will be significantly more complex in terms of decoding

and control circuitry. For example, an outer-loop ISA defining a general-purpose 64-bit processor would have many more instructions than an inner-loop ISA that is dedicated solely to data compression.

Referring now to Figure 12A, a diagram showing an exemplary allocation of
5   reconfigurable hardware resources between the IFU 60, the DOU 62, and the AOU 64 for an outer-loop ISA is shown. In the exemplary allocation of reconfigurable hardware resources for the outer-loop ISA, the IFU 60, the DOU 62, and the AOU 64 are each allocated approximately one-third of the available reconfigurable hardware resources. In the event that the DRPU 32 is to be reconfigured to implement an inner-loop ISA, fewer reconfigurable hardware resources are
10   required to implement the IFU 60 and the AOU 64 due to the limited number of instructions and types of address operations supported by an inner-loop ISA. Referring also now to Figure 12B, a diagram showing an exemplary allocation of reconfigurable hardware resources between the IFU 60, the DOU 62, and the AOU 64 for an inner-loop ISA is shown. In the exemplary allocation of reconfigurable hardware resources for the inner-loop ISA, the IFU 60 is implemented using
15   approximately 5 to 10 percent of the reconfigurable hardware resources, and the AOU 64 is implemented using approximately 10 to 25 percent of the reconfigurable hardware resources. Thus, approximately 70 to 80 percent of the reconfigurable hardware resources remain available for implementing the DOU 62. This in turn means that the internal structure of the DOU 62 associated with the inner-loop ISA can be more complex and therefore offer significantly higher
20   performance than the internal structure of the DOU 62 associated with the outer-loop ISA.

Those skilled in the art will recognize that the DRPU 32 may exclude either the DOU 62 or the AOU 64 in an alternate embodiment. For example, in an alternate embodiment the DRPU 32 may not include an AOU 64. The DOU 62 would then be responsible for performing operations upon both data and addresses. Regardless of the particular DRPU embodiment
25   considered, a finite number of reconfigurable hardware resources must be allocated to implement the elements of the DRPU 32. The reconfigurable hardware resources are preferably allocated such that optimum or near-optimum performance is achieved for the currently-considered ISA relative to the total space of available reconfigurable hardware resources.

Those skilled in the art will recognize that the detailed structure of each element of the
30   IFU 60, the DOU 62, and the AOU 64 is not limited to the embodiments described above. For a given ISA, the corresponding configuration data set is preferably defined such that the internal structure of each element within the IFU 60, the DOU 62, and the AOU 64 maximizes computational performance relative to the available reconfigurable hardware resources.

Referring now to Figure 13, a block diagram of a preferred embodiment of a T-machine
35   14 is shown. The T-machine 14 comprises a second local time-base unit 300, a common interface and control unit 302, and a set of interconnect I/O units 304. The second local time-

base unit 300 has a timing input that forms the T-machine's master timing input. The common interface and control unit 302 has a timing input coupled to a timing output of the second local time-base unit 300 via a second timing signal line 310, an address output coupled to the address line 44, a first bidirectional data port coupled to the memory I/O line 46, a bidirectional control

5   port coupled to the external control line 48, and a second bidirectional data port coupled to a bidirectional data port of each interconnect I/O unit 304 present via a message transfer line 312. Each interconnect I/O unit 304 has an input coupled to the GPIM 16 via a message input line 314, and an output coupled to the GPIM 16 via a message output line 316.

The second local time-base unit 300 within the T-machine 14 receives the master timing

10   signal from the master time-base unit 22, and generates a second local timing signal. The second local time-base unit 300 delivers the second local timing signal to the common interface and control unit 302, thereby providing a timing reference for the T-machine 14 in which it resides. Preferably, the second local timing signal is phase-synchronized with the master timing signal. Within the system 10, each T-machine's second local time-base unit 300 preferably operates at an

15   identical frequency. Those skilled in the art will recognize that in an alternate embodiment, one or more second local time-base units 300 could operate at different frequencies. The second local time-base unit 300 is preferably implemented using conventional phase-locked frequency-. conversion circuitry, including CLB-based phase-lock detection circuitry. Those skilled in the art will recognize that in an alternate embodiment, the second local time-base unit 300 could be

20   implemented as a portion of a clock distribution tree.

The common interface and control unit 302 directs the transfer of messages between its corresponding S-machine 12 and a specified interconnect I/O unit 304, where a message includes a command and possibly data. In the preferred embodiment, the specified interconnect I/O unit 304 may reside within any T-machine 14 or I/O T-machine 18 internal or external to the system

25   10. In the present invention, each interconnect I/O unit 304 is preferably assigned an interconnect address that uniquely identifies the interconnect I/O unit 304. The interconnect addresses for the interconnect I/O units 304 within a given T-machine are stored in the corresponding S-machine's architecture description memory 101.

The common interface and control unit 302 receives data and commands from its

30   corresponding S-machine 12 via the memory I/O line 46 and the external control signal line 48, respectively. Preferably, each command received includes a target interconnect address and a command code that specifies a particular type of operation to be performed. In the preferred embodiment, the types of operations uniquely identified by command codes include: 1) data read operations; 2) data write operations; and 3) interrupt signal transfer, including reconfiguration

35   interrupt transfer. The target interconnect address identifies a target interconnect I/O unit 304 to which data and commands are to be transferred. Preferably, the common interface and control

unit 302 transfers each command and any related data as a set of packet-based messages in a conventional manner, where each message includes the target interconnect address and the command code.

In addition to receiving data and commands from its corresponding S-machine 12, the

5   common interface and control unit 302 receives messages from each of the interconnect I/O units 304 coupled to the message transfer line 312. In the preferred embodiment, the common interface and control unit 302 converts a group of related messages into a single command and data sequence. If the command is directed to the DRPU 32 within its corresponding S-machine 12, the common interface and control unit 302 issues the command via the external control signal

10   line 48. If the command is directed to the memory 34 within its corresponding S-machine 12, the common interface and control unit 302 issues an appropriate memory control signal via the external control signal line 48 and a memory address signal via the memory address line 44. Data is transferred via the memory I/O line 46. In the preferred embodiment, the common interface and control unit 302 comprises CLB-based circuitry to implement operations analogous

15   to those performed by a conventional SCI switching unit as defined by ANSI/IEEE Standard 1596-1992.

Each interconnect I/O unit 304 receives messages from the common interface and control unit 302, and transfers messages to other interconnect I/O units 304 via the GPIM 16, under direction of control signals received from the common interface and control unit 302. In the

20   preferred embodiment, the interconnect I/O unit 304 is based upon an SCI node as defined by ANSI/IEEE Standard 1596-1992. Referring now to Figure 14, a block diagram of a preferred embodiment of an interconnect I/O unit 304 is shown. The interconnect I/O unit 304 comprises an address decoder 320, an input FIFO buffer 322, a bypass FIFO buffer 324, an output FIFO buffer 326, and a multiplexor 328. The address decoder 320 has an input that forms the

25   interconnect I/O unit's input, a first output coupled to the input FIFO 322, and a second output coupled to the bypass FIFO 324. The input FIFO 322 has an output coupled to the message transfer line 312 for transferring messages to the common interface and control unit 302. The output FIFO 326 has an input coupled to the message transfer line 312 for receiving messages from the common interface and control unit 302, and an output coupled to a first input of the

30   multiplexor 328. The bypass FIFO 326 has an output coupled to a second input of the multiplexor 328. Finally, the multiplexor 328 has a control input coupled to the message transfer line 312, and an output that forms the interconnect I/O unit's output.

The interconnect I/O unit 304 receives messages at the input of the address decoder 320. The address decoder 320 determines whether the target interconnect address specified in a

35   received message is identical to the interconnect address of the interconnect I/O unit 304 in which it resides. If so, the address decoder 320 routes the message to the input FIFO 322.

Otherwise, the address decoder 320 routes the message to the bypass FIFO 324. In the preferred embodiment, the address decoder 320 comprises a decoder and a data selector implemented using IOBs and CLBs.

The input FIFO 322 is a conventional FIFO buffer that transfers messages received at its
5 input to the message transfer line 312. Both the bypass FIFO 324 and the output FIFO 326 are conventional FIFO buffers that transfer messages received at their inputs to the multiplexor 328. The multiplexor 328 is a conventional multiplexor that routes either a message received from the bypass FIFO 324 or a message received from the output FIFO 326 to the GPIM 16 in accordance with a control signal received at its control input. In the preferred embodiment, each of the input
10 FIFO 322, the bypass FIFO 324, and the output FIFO 326 are implemented using a set of CLBs. The multiplexor 328 is preferably implemented using a set of CLBs and IOBs.

Referring now to Figure 15, a block diagram of a preferred embodiment of an I/O T-machine 18 is shown. The I/O T-machine 18 comprises a third local time-base unit 360, a common custom interface and control unit 362, and an interconnect I/O unit 304. The third local
15 time-base unit 360 has a timing input that forms the I/O T-machine's master timing input. The interconnect I/O unit 304 has an input coupled to the GPIM 16 via a message input line 314, and an output coupled to the GPIM 16 via a message output line 316. The common custom interface and control unit 362 preferably has a timing input coupled to a timing output of the third local time-base unit 360 via a third timing signal line 370, a first bidirectional data port coupled to a
20 bidirectional data port of the interconnect I/O unit 304, and a set of couplings to an I/O device 20. In the preferred embodiment, the set of couplings to the I/O device 20 includes a second bidirectional data port coupled to a bidirectional data port of the I/O device 20, an address output coupled to an address input of the I/O device 20, and a bidirectional control port coupled to a bidirectional control port of the I/O device 20. Those skilled in the art will readily recognize that
25 the couplings to the I/O device 20 are dependent upon the type of I/O device 20 to which the common custom interface and control unit 362 is coupled.

The third local time-base unit 360 receives the master timing signal from the master time-base unit 22, and generates a third local timing signal. The third local time-base unit 360 delivers the third local timing signal to the common custom interface and control unit 362, thus
30 providing a timing reference for the I/O T-machine in which it resides. In the preferred embodiment, the third local timing signal is phase-synchronized with the master timing signal. Each I/O T-machine's third local time-base unit 360 preferably operates at an identical frequency. In an alternate embodiment, one or more third local time-base units 360 could operate at different frequencies. The third local time-base unit 360 is preferably implemented using conventional
35 phase-locked frequency-conversion circuitry that includes CLB-based phase-lock detection circuitry. In a manner analogous to that for the first and second local time-base units 30, 300, the

third local time-base unit 360 could be implemented as a portion of a clock distribution tree in an alternate embodiment.

The structure and functionality of the interconnect I/O unit 304 within the I/O T-machine 18 is preferably identical to that previously described for the T-machine 14. The interconnect
5   I/O unit 304 within the I/O T-machine 18 is assigned a unique interconnect address in a manner analogous to that for each interconnect I/O unit 304 within any given T-machine 14.

The common custom interface and control unit 362 directs the transfer of messages between the I/O device 20 to which it is coupled and the interconnect I/O unit 304, where a message includes a command and possibly data. The common custom interface and control unit
10   362 receives data and commands from its corresponding I/O device 20. Preferably, each command received from the I/O device 20 includes a target interconnect address and a command code that specifies a particular type of operation to be performed. In the preferred embodiment, the types of operations uniquely identified by command codes include: 1) data requests; 2) data transfer acknowledgments; and 3) interrupt signal transfer. The target interconnect address
15   identifies a target interconnect I/O unit 304 in the system 10 to which data and commands are to be transferred. Preferably, the common interface and control unit 362 transfers each command and any related data as a set of packet-based messages in a conventional manner, where each message includes the target interconnect address and the command code.

In addition to receiving data and commands from its corresponding I/O device 20, the
20   common custom interface and control unit 362 receives messages from its associated interconnect I/O unit 304. In the preferred embodiment, the common custom interface and control unit 362 converts a group of related messages into a single command and data sequence in accordance with the communication protocols supported by its corresponding I/O device 20. In the preferred embodiment, the common custom interface and control unit 362 comprises a
25   CLB-based I/O device controller coupled to CLB-based circuitry for implementing operations analogous to those performed by a conventional SCI switching unit as defined by ANSI/IEEE Standard 1596-1992.

The GPIM 16 is a conventional interconnect mesh that facilitates point-to-point parallel message routing between interconnect I/O units 304. In the preferred embodiment, the GPIM 16
30   is a wire-based k-ary n-cube static interconnect network. Referring now to Figure 16, a block diagram of an exemplary embodiment of a General Purpose Interconnect Matrix 16 is shown. In Figure 16, the GPIM 16 is a toroidal interconnect mesh, or equivalently, a k-ary 2-cube, comprising a plurality of first communication channels 380 and a plurality of second communication channels 382. Each first communication channel 380 includes a plurality of
35   node connection sites 384, as does each second communication channel 382. Each interconnect I/O unit 304 in the system 10 is preferably coupled to the GPIM 16 such that the message input

line 314 and the message output line 316 join consecutive node connection sites 384 within a given communication channel 380, 382. In the preferred embodiment, each T-machine 14 includes an interconnect I/O unit 304 coupled to the first communication channel 380 and an interconnect I/O unit 304 coupled to the second communication channel 382 in the manner

5   described above. The common interface and control unit 302 within the T-machine 14 preferably facilitates the routing of information between its interconnect I/O unit 304 coupled to the first communication channel and its interconnect I/O unit 304 coupled to the second communication channel 382. Thus, for a T-machine 14 having an interconnect I/O unit 304 coupled to the first communication channel labeled as 380c and an interconnect I/O unit 304 coupled to the second

10   communication channel labeled as 382c in Figure 16, this T-machine's common interface and control unit 302 facilitates information routing between this set of first and second communication channels 380c, 382c.

The GPIM 16 thus facilitates the routing of multiple messages between interconnect I/O units 304 in parallel. For the two-dimensional GPIM 16 shown in Figure 16, each T-machine 14

15   preferably includes a single interconnect I/O unit 304 for the first communication channel 380 and a single interconnect I/O unit 304 for the second communication channel 382. Those skilled in the art will recognize that in an embodiment in which the GPIM 16 has a dimensionality greater than two, the T-machine 14 preferably includes more than two interconnect I/O units 304. Preferably, the GPIM 16 is implemented as a k-ary 2-cube having a 16-bit datapath size.

20   In the preceding description, various elements of the present invention are preferably implemented using reconfigurable hardware resources. The manufacturers of reprogrammable logic devices typically provide published guidelines for implementing conventional digital hardware using reprogrammable or reconfigurable hardware resources. For example, the 1994 Xilinx Programmable Logic Data Book (Xilinx, Inc., San Jose, CA) includes Application Notes

25   such as the following: Application Note XAPP 005.002, "Register-Based FIFO"; Application Note XAPP 044.00 "High-Performance RAM-Based FIFO"; Application Note XAPP 013.001, "Using the Dedicated Carry Logic in the XC4000"; Application Note XAPP 018.000, "Estimating the Performance of XC4000 Adders and Counters"; Application Note XAPP 028.001, "Frequency/Phase Comparator for Phase-Locked Loops"; Application Note XAPP

30   031.000, "Using the XC4000 RAM Capability"; Application Note XAPP 036.001, "Four-Port DRAM Controller..."; and Application Note XAPP 039.001, "18-Bit Pipelined Accumulator." Additional material published by Xilinx includes features in "XCELL, The Quarterly Journal for Xilinx Programmable Logic Users." For example, an article detailing the implementation of fast integer multipliers appears in Issue 14, the Third Quarter 1994 issue.

35   The system 10 described herein is a scalable, parallel computer architecture for dynamically implementing multiple ISAs. Any individual S-machine 12 is capable of running an

-41-

entire computer program by itself, independent of another S-machine 12 or external hardware resources such as a host computer. On any individual S-machine 12, multiple ISAs are implemented sequentially in time during program execution in response to reconfiguration interrupts and/or program-embedded reconfiguration directives. Because the system 10

5   preferably includes multiple S-machines 12, multiple programs are preferably executed simultaneously, where each program may be independent. Thus, because the system 10 preferably includes multiple S-machines 12, multiple ISAs are implemented simultaneously (i.e., in parallel) at all times other than during system initialization or reconfiguration. That is, at any given time, multiple sets of program instructions are executed simultaneously, where each set of

10   program instructions is executed according to a corresponding ISA. Each such ISA may be unique.

S-machines 12 communicate with each other and with I/O devices 20 via the set of T-machines 14, the GPIM 16, and each I/O T-machine 18. While each S-machine 12 is an entire computer in itself that is capable of independent operation, any S-machine 12 is capable of

15   functioning as a master S-machine 12 for other S-machines 12 or the entire system 10, sending data and/or commands to other S-machines 12, one or more T-machines 16, one or more I/O T-machines 18, and one or more I/O devices 22.

The system 10 of the present invention is thus particularly useful for problems that can be divided both spatially and temporally into one or more data-parallel subproblems, for example:

20   image processing, medical data processing, calibrated color matching, database computation, document processing, associative search engines, and network servers. For computational problems with a large array of operands, data parallelism exists when algorithms can be applied so as to offer an effective computational speed-up through parallel computing techniques. Data parallel problems possess known complexity, namely, $O(n^k)$. The value of k is problem-

25   dependent; for example, k = 2 for image processing, and k = 3 for medical data processing. In the present invention, individual S-machines 12 are preferably utilized to exploit data parallelism at the level of program instruction groups. Because the system 10 includes multiple S-machines 12, the system 10 is preferably utilized to exploit data parallelism at the level of sets of entire programs.

30   The system 10 of the present invention provides a great deal of computational power because of its ability to completely reconfigure the instruction processing hardware in each S-machine 12 to optimize the computational capabilities of such hardware relative to computational needs at any given moment. Each S-machine 12 can be reconfigured independently of any other S-machine 12. The system 10 advantageously treats each configuration data set, and hence each

35   ISA, as a programmed boundary or interface between software and the reconfigurable hardware described herein. The architecture of the present invention additionally facilitates the high-level

structuring of reconfigurable hardware to selectively address the concerns of actual systems in situ, including: manners in which interruption affect instruction processing; the need for deterministic latency response to facilitate real-time processing and control capabilities; and the need for selectable responses to fault-handling.

5   In contrast with other computer architectures, the present invention teaches the maximal utilization of Silicon resources at all times. The present invention provides for a parallel computer system that can be increased to any desired size at any time, even to massively parallel sizes comprising thousands of S-machines 12. Such architectural scalability is possible because S-machine-based instruction processing is intentionally separated from T-machine-based data

10   communication. This instruction processing/data communication separation paradigm is extremely well-suited for data-parallel computation. The internal structure of S-machine hardware is preferably optimized for time-flow of instructions, while the internal structure of T-machine hardware is preferably optimized for efficient data communication. The set of S-machines 12 and the set of T-machines are each a separable, configurable component in a space-

15   time division of data-parallel computational labor.

With the present invention, future reconfigurable hardware may be exploited to construct systems having ever-greater computational capabilities while maintaining the overall structure described herein. In other words, the system 10 of the present invention is technologically scalable. Virtually all current reconfigurable logic devices are memory-based Complementary

20   Metal-Oxide Semiconductor (CMOS) technology. Advances in device capacity follow semiconductor memory technology trends. In future systems, a reconfigurable logic device used to construct an S-machine 12 would have a division of internal hardware resources in accordance with the inner-loop and outer-loop ISA parametrics described herein. Larger reconfigurable logic devices simply offer the capability to perform more data parallel computational labor

25   within a single device. For example, a larger functional unit 194 within the second exemplary embodiment of the DOU 63 as described above with reference to Figure 9B would accommodate larger imaging kernel sizes. Those skilled in the art will recognize that the technological scalability provided by the present invention is not limited to CMOS-based devices, nor is it limited to FPGA-based implementations. Thus, the present invention provides technological

30   scalability regardless of the particular technology used to provide reconfigurability or reprogrammability.

Referring now to Figures 17A and 17B, a flowchart of a preferred method for scalable, parallel, dynamically reconfigurable computing is shown. Preferably, the method of Figures 17A and 17B is performed within each S-machine 12 in the system 10. The preferred method begins

35   in step 1000 of Figure 17A with the reconfiguration logic 104 retrieving a configuration data set corresponding to an ISA. Next, in step 1002, the reconfiguration logic 104 configures each

element within the IFU 60, the DOU 62, and the AOU 64 according to the retrieved configuration data set in step 1002, thereby producing a DRPU hardware organization for the implementation of the ISA currently under consideration. Following step 1002, the interrupt logic 106 retrieves the interrupt response signals stored in the architecture description memory

5   101, and generates a corresponding set of transition control signals that define how the current DRPU configuration responds to interrupts in step 1004. The ISS 100 subsequently initializes program state information in step 1006, after which the ISS 100 initiates an instruction execution cycle in step 1008.

Next, in step 1010, the ISS 100 or the interrupt logic 106 determines whether

10   reconfiguration is required. The ISS 100 determines that reconfiguration is required in the event that a reconfiguration directive is selected during program execution. The interrupt logic 106 determines that reconfiguration is required in response to a reconfiguration interrupt. If reconfiguration is required, the preferred method proceeds to step 1012, in which a reconfiguration handler saves program state information. Preferably, the program state

15   information includes a reference to the configuration data set corresponding to the current DRPU configuration. After step 1012, the preferred method returns to step 1000 to retrieve a next configuration data set as referenced by the reconfiguration directive or the reconfiguration interrupt.

In the event that reconfiguration is not required in step 1010, the interrupt logic 106

20   determines whether a non-reconfiguration interrupt requires servicing in step 1014. If so, the ISS 100 next determines in step 1020 whether a state transition from the current ISS state within the instruction execution cycle to the interrupt service state is allowable based upon the transition control signals. If a state transition to the interrupt service state is not allowed, the ISS 100 advances to a next state in the instruction execution cycle, and returns to state 1020. In the event

25   that the transition control signals allow a state transition from the current ISS state within the instruction execution cycle to the interrupt service state, the ISS 100 next advances to the interrupt service state in step 1024. In step 1024, the ISS 100 saves program state information and executes program instructions for servicing the interrupt. Following step 1024, the preferred method returns to step 1008 to resume the current instruction execution cycle if it had not been

30   completed, or to initiate a next instruction execution cycle.

In the event that no non-reconfiguration interrupt requires servicing in step 1014, the preferred method proceeds to step 1016 and determines whether execution of the current program is complete. If execution of the current program is to continue, the preferred method returns to step 1008 to initiate another instruction execution cycle. Otherwise, the preferred method ends.

35   The present invention also incorporates a meta-addressing mechanism for performing the memory operations required by the architecture of the present invention. In accordance with the

present invention, the T-machines 14 are used as addressing machines. The T-machines 14 perform interrupt handling, queuing of messages, meta-address generation, and control the overall transfer of data packets. Figure 18 illustrates a data packet 1800 in accordance with the present invention. The data packet 1800 comprises a data portion 1824, a command portion

5   1820, a source geographic address 1816, a size delimiter 1812, a target local address 1808, and a target geographic address 1804. A meta-address 1828 includes the target geographic address 1804 and the target local memory address 1808. The target local address 1808 specifies where in the local memory 34 the data of the data packet 1800 should be written. The target geographic or interconnect address 1804 specifies which T-machine 14 should receive the data packet 1800.

10   The source geographic address 1816 specifies the T-machine 14 which originated the data packet 1800.

Any two source and destination pair geographic addresses 1816, 1804 uniquely determine one pathway to a local address space of 264 bits. However, there can be more than one of these pathways in a system, and these pathways can operate in parallel. An S-machine 12 can have

15   any number supporting T-machines 14 coupled to it, up to the local memory bandwidth and in consideration of queuing effects. Thus, in addition to allowing irregular power-of-two scalability, and in addition to allowing non-uniform processors in the system, the present invention also allows arbitrary scalability of the number of unique pathways to each S-machine 12. This type of scalability is important in many applications, such as in distributed image

20   processing, where a pyramid or tree of dynamically reconfigurable processing elements might be devised to enable more communication bandwidth to be provided to higher levels of the system. If desired, this pyramid architecture is implemented by allocating more of the uniform-speed T-machines 14 to be accessible to higher levels of the pyramid of S-machines 12, providing the addressing power to the S-machines 12 which require it most. This provides a more cost-

25   effective system as system resources can be devoted to the most processing and communication intensive tasks.

In a preferred embodiment, the meta-address is eighty bits wide. In this embodiment, the geographic address is sixteen bits and the local memory address is sixty-four bits wide. The sixteen bit geographic address allows 65536 individual geographic addresses to be specified.

30   The sixty-four bit local memory address allows 2^{64} separate addressable bits within each local memory 34 to be specified. Each S-machine 12 may have a local memory 34 which is configured for the specific S-machine 12. As the S-machines 12 and their memories 34 are isolated from each other, there is no requirement of uniformity of size or structure of the memories, or maintenance of coherency or consistency across the memories. As long as the program

35   instructions of the source S-machine 12 are written in awareness of the architecture of the local memory 34 of the target S-machine 12 and correctly specify the memory location, the local

memory 34 of the target S-machine 12 is easily and readily addressed regardless of its size and layout. This modularity allows the present architecture to be scaled up or down in size using a variety of components without regard for addressing concerns. Integration of new S-machines is greatly simplified as well. If a new S-machine 12 is added to the system, a new geographic

5   address is selected for the S-machine 12, and programs requiring the use of the new S-machine 12 are given the new address. Once the new address is incorporated into the programs designed to take advantage of the new S-machine 12, there are no other conflicts to resolve or calculations to perform; the S-machine 12 is integrated.

Figure 19 illustrates the processing of the S-machine 12 of the present invention for

10   requesting a remote operation. The S-machine 12 receives 1900 instructions. The S-machine 12 determines 1904 whether the instruction requires a remote operation. If the instruction does not require a remote operation, the instruction is executed 1916. If the instruction does require a remote operation, the remote operation information is stored 1904 into the local memory. The S-machine 12 determines an instruction requires a remote operation by examining the status of a

15   flag in the instruction code which indicates whether a remote operation is required after which the method proceeds to step 1920 as will be described below. A remote operation is a operation which requires the use of a different S-machine 12 in order to achieve a result. Remote operation information is provided by the program being executed by an S-machine 12 and is stored into local memory 34 when a remote operation is desired. A consistent memory location in local

20   memory 34 is preferably used to store the remote operation information to allow the T-machine 14 to immediately access the information without having to first obtain an address. Remote operation information typically includes the target geographic address 1804 of the remote T-machine 14, the target local memory address 1808 to store data to or retrieve data from the remote S-machine 12, command information 1820, size information 1812, and data 1824. All of

25   this information is stored into the local memory 34 by the S-machine 12 upon determining that the instruction requires a remote operation.

In one embodiment, the S-machine 12 issues 1912 an imperative to the T-machine to indicate that a remote operation is needed. An imperative is a unique command string which the T-machines 14 are designed to recognize. An imperative typically consists of a memory address

30   where the remote operation information is located in local memory 34, and a size delimiter to indicate the size of the addressing information. Multiple remote operations can be requested at a single time by the program being executed by the S-machine 12 by simply specifying a beginning address for the remote operation information and a series of size delimiters. The T-machine 14 is able to then process the different requests for information sequentially. The S-

35   machine 12 then determines 1920 whether there are any other instructions to be performed. If there are, the next instruction is received and executed. Thus, the S-machine 12 is able to almost

instantaneously continue the execution of instructions despite the requirement of remote operations. As the T-machine 14 performs the transfer and retrieval of data, the processing power of the S-machine 12 is freed to exclusively focus on processing instructions.

Figure 20 illustrates the processing of the T-machines 14 in receiving an imperative from an S-
5   machine 12. First, the T-machine 14 determines 2000 whether a command received on control line 48 from the S-machine 12 is an imperative. Responsive to determining a command is an imperative, the T-machine 14 retrieves 2004 remote operation information through memory/data line 46 from the local memory 34. The remote operation information is preferably located in a consistent location in memory 34 in order to allow the T-machine 14 to retrieve the data without
10   having to determine a new memory address each time remote information is to be retrieved. Alternatively, the remote operation information can be stored in random places in local memory 34; however, the location of the information must then be transmitted as a part of the imperative. After retrieving the remote operation information, the T-machine 14, specifically, the CICU 302 component of the T-machine 14, generates 208 a meta-address 1828 from the information. The
15   target local address 1808 is appended to the target geographic address 1804 to form the meta-address 1828. The T-machine 14 then generates 2112 the data packet 1800 from the remaining remote operation information, and transmits the data packet 1800 to the interconnect unit or GPIM 16 for transmission to the destination as required.

The source geographic address 1816 may be specified by the program instructions and
20   thus stored into local memory 34 for retrieval by the T-machine 14, or the source geographic address 1816 is preferably stored in an architecture description memory (ADM) 101. The ADM 101 is a modifiable memory which stores a geographic address for the T-machine 14 to which it is coupled. By using an ADM 101, the geographic addresses of the entire system may be changed transparently. In this embodiment of the system, the T-machine 14 retrieves the source
25   geographic address 1816 from the ADM 101 to ensure that it is using the most current version of its own source geographic address 1816. In an embodiment where multiple CICUs 302 are coupled to each S-machine 12, the geographic address for each CICU 302 is stored in the ADM 101.

Figure 21 illustrates the processing of the T-machine 14 for receiving data packets which
30   have been transmitted through the interconnect unit. The T-machine 14 receives 2100 a data packet from the interconnect unit. The T-machine 14 decodes 2104 the data packet 1800 by parsing out the target geographic address 1804 component of the meta-address 1828. As described above, the address decoder 320 of the T-machine 14 decodes the data packet 1800. The address decoder 320 compares 2108 the geographic address 1804 with an associated
35   geographic address. In an embodiment which uses modifiable ADMs 101, the address decoder 320 compares the received geographic address 1804 with the address stored in the ADM 101. If

the address decoder 320 determines 2012 that the geographic addresses match, the data packet 1800 is transmitted to local memory 34 to the location specified by the local memory address 1808. The data packet 1800 is parsed and the data is sent over memory/data line 46, and the command is sent over control line 48. The address information is sent over address line 44. If

5   the addresses do not match, an error message is transmitted to the T-machine 14 identified by the source geographic address 1816 component of the data packet 1800 through bypass FIFO 324, MUX 328, and GPIM 16, using the same process as when an misaddressed data packet 1800 is received by the T-machine 14, as described above. If the CICU 304 is currently assembling or deconstructing data packets 1800 when a new data packet 1800 is received, the T-machine 14

10   queues the data packet 1800 into the input FIFO 322 until such time as the CICU 304 is available to receive and process the data.

In an alternate embodiment, the T-machine 14 is equipped to recognize priorities of messages, and interrupt the processing of the S-machine 12 if appropriate to have the S-machine process the new command. In this embodiment, as illustrated in Figure 22, the CICU 302 has

15   additional components, including interrupt logic 2200, a comparator 2204, and a recognition unit 2208. Figure 23 illustrates the operation of the interrupt handling capabilities of the CICU 302. The recognition unit 2208 parses 2300 the data packet 1800 to identify the command 1820 after the address has been verified by the address decoder 320. The recognition unit 2208 determines 2304 whether the command 1820 is an interrupt request. If a command 1800 is an interrupt

20   request, the command 1820 will have an interrupt ID. If the command 1820 does not have an interrupt ID, the data packet is passed 2308 to the Common Interface and Control Unit 302 for processing as described above.

If the command 1820 does have an interrupt ID, the interrupt ID is passed to comparator 2204, which is also coupled to memory 34. Memory 34 stores a list of interrupt IDs. Each S-

25   machine 12 preferably has a list of interrupts which the S-machine 12 is designed to service stored in its associated local memory 34. This list identifies the interrupts and may specify a priority of the interrupts and contains instructions for executing the interrupts. The comparator 2204 compares 2312 the interrupt ID in the received command to the list of stored IDs. If the interrupt ID specified by the command does not match an ID in the list, an error message is

30   transmitted 2320 to the destination specified by the source geographic address 1816 through bypass FIFO 324, MUX 328, and to GPIM 16 across signal line 314. If the interrupt ID does match a stored ID, the interrupt logic 2200 processes 2324 the interrupt according to the information provided either in local memory 34 associated with the stored ID, or in accordance with the information provided in the data packet 1800, and communicates the resulting

35   commands to the S-machine 12 over control line 48.

If prioritization is enabled, the interrupt logic 2200 compares the priority of the interrupt request with the priorities of any data packets 1800 which are currently in the input FIFO 322. If the interrupt request has a higher priority than a data packet 1800 in the FIFO 322, the interrupt request is placed ahead of the lower priority data packet 1800. In some cases, the interrupt

5   request may require the S-machine 12 to stop executing. In this situation, a priority level is assigned to process executing in the S-machine 12. If the interrupt request has a priority greater than the priority of the currently executing process, the interrupt logic 2200 issues an imperative on control line 48 to the S-machine 12 to have the S-machine 12 cease execution of the current process and begin handling the interrupt request. Thus, a complete prioritization and interrupt

10   handling scheme is implemented by the T-machine 14 in accordance with the architecture of the present invention which requires minimal additional processing by the S-machine 12.

Thus, as the T-machine 14 performs all of the memory operation functions required by the computer system, the S-machine 12 is able to execute the main instructions of the program. The space-time separation of the memory and instruction execution operations greatly optimizes

15   the processing power of the multi-processor, highly-parallel system. As no virtual or shared memory is used, hardware consistency and coherency operations is not required. The S-machines 12 can operate at different rates, and the ISAs realized by dynamically reconfigurable S-machines 12 can be different. Further, the FPGAs which implement the S-machines 12 can also be optimized for a particular task. For example, in an embedded image-computing

20   environment, it is unnecessary to have a front-panel LCD screen controller be an imaging optimized S-machine 12. However, it is still very desirable to have all S-machines 12 in the system consistently addressable by each S-machines 12 which needs to communicate with another S-machine 12, and this is provided for by the present invention as described above. Software is used to provide cross-system coherency and consistency, using conventional methods

25   such as a Message Passing Interface (MPI) runtime library for the S-machines 12 and T-machines 14 or a runtime library for the Parallel Virtual Machine (PVM). Either MPI or PVM operate in effect as hardware abstraction layers (HALs). In accordance with the present invention, the HALs are for dynamically reconfigurable S-machines 12 and fixed T-machines 14. As the memory operations are entirely controlled by software, the system is dynamically

30 · reconfigurable, and is not subject to complicated hardware/software interactions. Thus, a completely scalable and architecturally reconfigurable computer system using independent and isolated memory and having separate addressing and processing machines is provided for use in a highly parallel computing environment. The use of the meta-address allows transparent and high granularity addressing, and allows the communication pathways of the computer system to

35 · be allocated and re-allocated as system requirements demand. The isolation of the addressing machines from the processing machines allows the processing machines to devote their resources

solely to processing, allows the processing machines to utilize divergent instruction set architectures, operate at different rates, and be implemented using individually optimized hardware, all of which greatly increase the processing power of the system.

The teachings of the present invention are distinctly different from other systems and
5 methods for reprogrammable or reconfigurable computing. In particular, the present invention is not equivalent to a downloadable microcode architecture, because such architectures rely upon a non-reconfigurable control means and non-reconfigurable hardware in general. The present invention is also distinctly different from an Attached Reconfigurable Processor (ARP) system, in which a set of reconfigurable hardware resources are coupled to a nonreconfigurable host
10 processor or host system. An ARP apparatus is dependent upon the host for executing some program instructions. Therefore, the set of available Silicon resources is not maximally utilized over the time frame of program execution because Silicon resources upon the ARP apparatus or the host will be idle or inefficiently used when the host or the ARP apparatus operates upon data, respectively. In contrast, each S-machine 12 is an independent computer in which entire
15 programs can be readily executed. Multiple S-machines 12 preferably execute programs simultaneously. The present invention therefore teaches the maximal utilization of Silicon resources at all times, for both single programs executing upon individual S-machines 12 and multiple programs executing upon the entire system 10.

An ARP apparatus provides a computational accelerator for a particular algorithm at a
20 particular time, and is implemented as a set of gates optimally interconnected with respect to this specific algorithm. The use of reconfigurable hardware resources for general-purpose operations such as managing instruction execution is avoided in ARP systems. Moreover, an ARP system does not treat a given set of interconnected gates as a readily reusable resource. In contrast, the present invention teaches a dynamically reconfigurable processing means configured for efficient
25 management of instruction execution, according to an instruction execution model best-suited to the computational needs at any particular moment. Each S-machine 12 includes a plurality of readily-reusable resources, for example, the ISS 100, the interrupt logic 106, and the store/align logic 152. The present invention teaches the use of reconfigurable logic resources at the level of groups of CLBs, IOBs, and reconfigurable interconnects rather than at the level of interconnected
30 gates. The present invention thus teaches the use of reconfigurable higher-level logic design constructs useful for performing operations upon entire classes of computational problems rather than teaching a single useful gate connection scheme useful for a single algorithm.

In general, ARP systems are directed toward translating a particular algorithm into a set of interconnected gates. Some ARP systems attempt to compile high-level instructions into an
35 optimal gate-level hardware configuration, which is in general an NP-hard problem. In contrast, the present invention teaches the use of a compiler for dynamically reconfigurable computing

that compiles high-level program instructions into assembly-language instructions according to a variable ISA in a very straightforward manner.

An ARP apparatus is generally incapable of treating its own host program as data or contextualizing itself. In contrast, each S-machine in the system 10 can treat its own programs as data, and thus readily contextualize itself. The system 10 can readily simulate itself through the execution of its own programs. The present invention additionally has the capability to compile its own compiler.

In the present invention, a single program may include a first group of instructions belonging to a first ISA, a second group of instructions belonging to a second ISA, a third group of instructions belonging to yet another ISA, and so on. The architecture taught herein executes each such group of instructions using hardware that is run-time configured to implement the ISA to which the instructions belong. No prior art systems or methods offer similar teachings.

The present invention further teaches a reconfigurable interruption scheme, in which interrupt latency, interrupt precision, and programmable state transition enabling may change according to the ISA currently under consideration. No analogous teachings are found in other computer systems. The present invention additionally teaches a computer system having a reconfigurable datapath bitwidth, address bitwidth, and reconfigurable control line widths, in contrast to prior art computer systems.

While the present invention has been described with reference to certain preferred embodiments, those skilled in the art will recognize that various modifications may be provided. Variations upon and modifications to the preferred embodiments are provided for by the present invention, which is limited only by the following claims.

APPENDIX A

5

INSTRUCTION SET 0,
### A GENERAL-PURPOSE OUTER-LOOP ISA

1.0 Programmer's Architectural Model

This section presents the programmers view of the ISAO architecture, including registers. memory model. calling conventions from high level languages. and interrupt model.

1.1 Registers

ISAO has 16 16-bit general purpose registers. 16 address registers. two processor status registers. and one interrupt vector register. The mnemonics for the data and address regis-
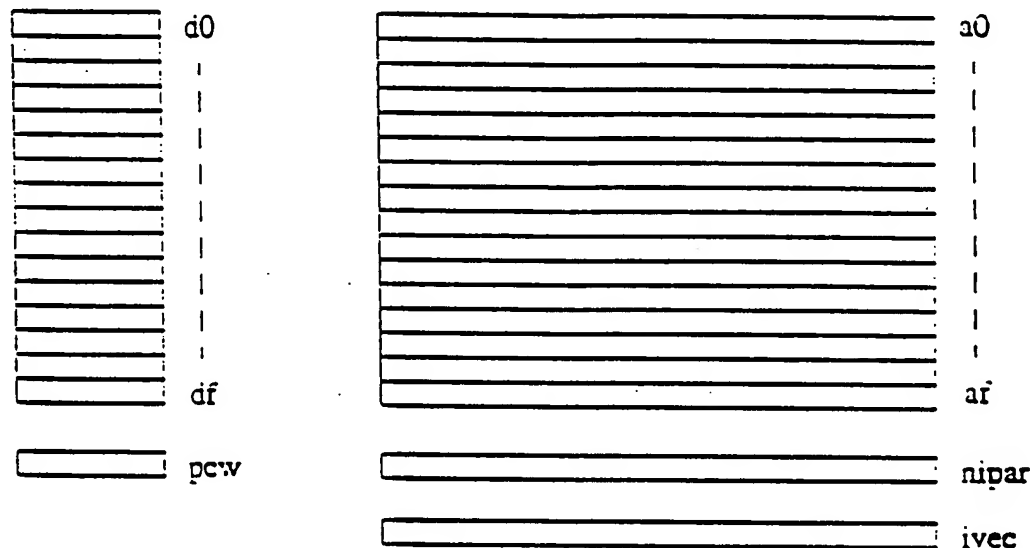
FIGURE 1. Registers

ters use hexadecimal number. therefore the last data register is df. and the last address register af. One of the processor status registers. nipar (Next Instruction Program Address Register), points to the address of the next instruction to fetch. The other status register. pcw (Processor Control Word), contains flags and control bits used to effect program flow and interrupt handling. It's bits are defined in Figure 2 on page 3. Undefined bits are reserved for future use. The four condition flags. Z. N. V, and C, are set as side effects of various instructions. See Section 2.0 for a summary of which flags are affected by each instruction.

The T (Trace Mode), and IM (Interrupt Mask) flag control how the processor responds to interrupts and when traps are handled. The interrupt vector register ivec holds the 64-bit address of the interrupt service routine. Interrupts and traps are described in Section 1.4 on page 3.

1.2 Memory access

Values that are stored in the 64-bit address registers are used by memory load/store instructions access memory in 16 and 64-bit increments (See Table 4 on page 7). The addresses are bit addresses. that is address 16 points to the word beginning at bit 16 in the

FIGURE 2. pcw fields

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
T							IM0				L	N	Z	V	C

Label	Meaning
T	Trace Mode
IM0	Interrupt Mask bit
N	Negative
Z	Zero
V	Overflow
C	Carry

memory. Words may only be read on 16-bit boundaries. and thus the four LSBs of an address register are ignored when reading memory. See [1] for further discussion of the concept of K_{isa}. 64-bit values are stored as 16-bit words in little-endian order (the least significant 16-bits is stored at the lowest address).

1.3 Calling conventions

By convention register af is used as the stack pointer by C programs. and register ae is used as the stack frame pointer. The mnemonics sp and fp may be used as aliases for these registers. All other registers are free for general use. The stack grows downward.

ints are 16-bits. longs are 64, as are void *s. int values are returned in d0. long and void * values in a0. d0-d4. and a0-a3 may be clobbered by functions. all other general registers must be preserved across function calls. Upon entry to a function the stack pointer points to the return address. and thus the first argument begins at address sp+64(decimal).

1.4 Traps and interrupts

ISA0 services one interrupt line. and software traps from two sources. All invoke the same flow-of-control transfer mechanism described below.

Externally, there is a single INTR signal input. and an iack output. iack goes active as soon as the interrupt mask bit in pcw is cleared by either by resetting pcw with an xpcw instruction. or restoring pcw to it's original value by returning from the interrupt with an rti instruction. The amount of time taken between signaling of the interrupt by the external device and servicing the interrupt by the processor is dependent on the instructions currently executing and the presence of software traps.

Software traps are triggered either by an explicit trap instruction. or by executing an instruction with the T (trace) flag set. In this case a control is transferred to the interrupt

service routine after the first instruction following the setting of the T. If a trap instruction is executed then the processor sets the T flag, and enters the interrupt service routine as though the T flag had been set before executing the instruction. No interrupts are serviced while the T flag is set. No more traps will occur until the T flag is cleared either by resetting pcw with an xpcw instruction. or resetting it from the stack by returning from the interrupt with an rti instruction.

Interrupts are caused by the presence of an active signal on the intr external signal. If the im flag or the T flag is set. then interrupts are masked and the pending interrupt is ignored. If the im flag and T flags are clear. then control is transferred to the interrupt service routine after the first instruction following the assertion of intr. Upon entry to the interrupt service routine the im flag is set by the processor. No more interrupts will occur until the im flag is cleared either by resetting pcw with an xpcw instruction. or resetting it from the stack by returning from the interrupt with an rti instruction.

The steps taken by the processor when an interrupt or trap occurs are:

1. All instructions currently executing are completed.

2. The contents of the 16 data registers (d0 first), the 16 address registers (a0 first), pcw. ivec. and nipar are pushed onto the stack (pointed to by register af) in this order. The value of af pushed onto the stack is it's value before the interrupt or trap servicing began.

3. If this is an interrupt. then interrupt bit in pcw is set to mask further interrupts. If this is a trap instruction then the T flag is set. If this is a trap caused by the T flag then pcw is not changed.

4. nipar is loaded with the value in the ivec register.

5. Execution of instructions in the interrupt handler then begins.

Upon execution of the rti instruction the following actions are taken:

1. The registers are restored from the stack in the opposite order in which they were written.

2. Execution resumes.

Note that if the interrupt mask flag had not already been cleared that it becomes cleared by the rti instruction since it was clear upon entry to the service routine. unless the value of pcw has been modified on the stack. If the T flag was set by executing a trap instruction then it is cleared upon execution of an rti for identical reasons. If the trap was caused by the T flag having been set prior to entry to the service routine then it must be cleared by the service routine to acknowledge that the trap has ocurred. When the interrupt mask flag is cleared by any means the external output signal iack becomes active for one clock cycle to signal the external device that the interrupt has been serviced.

2.0 Instructions grouped by function

The notational conventions are:

Notation	Meaning
d0, d1	Two data registers (which may be the same reg)
a0, a1	Two address registers (which may be the same reg)
<<	Left shift operation
>>	Right shift operation
K_n	N-bit constant
SK_n	Signed (twos complement) N-bit constant
(d1+1, d1)	A 32 value from a pair of data registers
(a0)	The value at the memory location pointed to by the register a0
K_{isa}	An architectural constant, which is 4 for ISA0. See reference [1]
AddrWidth	An architectural constant that denotes the bit width of an address register. Its value is 64

2.1 Register Movement

TABLE 1. Register Movement.

Mnemonic		Operation	Flags	Page
mov	d0, d1	d1 ← d0	Z,N	17
emov	a0, a1	a1 ← a0	Z,N	13
xda	d0, a0	a0 ← (d0+3, d0+2, d0+1, d0)	NONE	20
xad	a0, d0	(d0+3, d0+3, d0+1, d0) ← a0	NONE	20
xivec	a1	a1 ↔ ivec	NONE	21
xpcw	d0	d0 ↔ pcw	NONE	21

2.2 Logical Operations

TABLE 2. Logical Operations. Flags Modified: Z, N

Mnemonic		Operation	Flags	Page
and	d0, d1	d1 ← d1 * d0	Z,N	10
or	d0, d1	d1 ← d1 + d0	Z,N	17
xor	d0, d1	d1 ← d1 ⊕ d0	Z,N	20
mask	d0, d1	d1 ← d1 * ~d0	Z,N	16
inv	d0, d1	d1 ← ~d0	Z,N	14
sl	d0, d1	d1 ← d1 << d0	Z,N	13
rotl	d1	d1 ← d1 << 1	Z,N,C	17
ksl	K_4, d0	d1 ← d1 << K_4	Z,N	15
esl	a1	a1 ← a1 << 1	Z,N	13
erotl	a1	a1 ← a1 << 1 | C	Z,N,C	13
sr	d0, d1	d1 ← d1 >> d0	Z,N	19
ksr	K_4, d1	d1 ← d1 >> K_4	Z,N	16
esr	a1	a1 ← a1 >> 1	Z,N	14
byte	a0, d1	d1 ← (d1 >>($a0_3$*8)) *0xff	Z,N[a]	10

a. The N flag is set by bit 7, not bit 15.

2.3 Memory Load/Store

TABLE 3. Load/Store. Flags Modified: Z, N

Mnemonic		Operation	Page
st	d0, a1	(a1) ← d0	18
str	d0, a1	a1 ← a1 - (1<<K_{isa}) (a1) ← d0	18
estr	a0, a1	a1 ← a1 - (1<<K_{isa})*(AddrWidth/K_{isa}) (a1) ← a0	14
ld	a0, d1	d1 ← (a0)	16
ldf	a0, d1	d1 ← (a0) a1 ← a1 + (1<<K_{isa})	16
eldf	a0, a1	a1 ← (a0) a1 ← a1 + (1<<K_{isa})*(AddrWidth/K_{isa})	12
ldi	K_{16}, d0	d0 ← K_{16}	16
eldi	K_{64}, a0	a0 ← K_{64}	12

2.4 Arithmetic Operations

TABLE 4. Arithmetic Operations. Flags Modified: Z.N.V.C

Mnemonic		Operation	Page
add	d0, d1	d1 ← d1 - d0	9
addc	d0, d1	d1 ← d1 - d0 - c	9
addq	SKg, d1	d1 ← d1 - SKg	9
sub	d0, d1	d1 ← d1 - d0	13
subc	d0, d1	d1 ← d1 - d0 - c	13
mul	d0, d1	(d1-1,d1) ← d1 * d0	1?
umul	d0, d1	(d1-1,d1) ← d1 * d0	13
div	d0, d1	(d1+1,d1) ← (d1+1,d1)/d0	12
udiv	d0, d1	(d1-1,d1) ← (d1+1,d1)/d0	13
eadd	d0, a1	a1 ← a1 - d0	11
efadd	a0, a1	a1 ← a1 - a0	12
eaddq	SKg, a1	a1 ← a1 - SKg	11
esub	d0, a1	a1 ← a1 - d0	14
efsub	a0, a1	a1 ← a1 - a0	12
cmp	d0, d1	∅ ← d1 - d0	11
ecmp	a0, a1	∅ ← a1 - a0	12

2.5   Control Flow

TABLE 5. Control Flow. Flags Modified: NONE

Mnemonic		Operation	Page
jmp	addr$_{64}$	nipar $\leftarrow$ addr$_{64}$	15
jcc	addr$_{64}$	(CC==1)? nipar $\leftarrow$ addr$_{64}$: nipar $\leftarrow$ nipar+(1<<K$_{isa}$)	15
brc	CC, SK$_{16}$	(CC==1)? nipar $\leftarrow$ nipar - SK$_{16}$ nipar $\leftarrow$ nipar+(1<<K$_{isa}$)	10
jsr	a0, a1	a1 $\leftarrow$ a1 - (1<<K$_{isa}$)*(AddrWidth/K$_{isa}$); (a1) $\leftarrow$ nipar nipar $\leftarrow$ a0	15
rts	a1	nipar $\leftarrow$ (a1); a1 $\leftarrow$ a1 - (1<<K$_{isa}$)*(AddrWidth/K$_{isa}$)	13
tu	a0	See Section 1.4	19
rti	a1	See Section 1.4	17

3.0 Alphabetical reference

The instructions set for ISA0 is listed below in alphabetical order. The mnemonic is given with a brief description. Below that is the binary coding of the instruction. Each line in the binary coding is a 16-bit word. The affected flags are then listed. Unless otherwise specified the flags are set using the data stored in the destination register. It is assumed that nipar has already been incremented at the beginning of instruction execution. Finally a text description of the instruction semantics is provided.

The notational conventions used in the binary codings are described in the following table. The condition codes are defined in Table 7 on page 22.

TABLE 6. Notational Conventions

Notation	Meaning
dddd_d	Destination data register
dddd_s	Source data register
aaaa_d	Destination address register
aaaa_s	Source address register
cccc	Condition code
xxxx	Unsigned 4 bit constant
sxxxxxxx	Signed 8 bit constant
sxxxxxxxxxxxxxxx	Signed 16-bit constant
xxxxxxxxxxxxxxxx	Unsigned 16-bit constant

add - Add data registers

| 1110 | dddd_d | 0000 | dddd_s |

Flags: Z,N,V,C

Adds two data registers, leaving the result in the destination register.

addc - Add data registers with carry

| 1110 | dddd_d | 0010 | dddd_s |

Flags: Z,N,V,C

Adds two data registers plus the carry flag, leaving the result in the destination register.

addq - Add quick constant

1100	dddd₄	sxxxxxxx

Flags: Z,N,V,C

Adds an 8 bit signed (twos complement) constant to a data register, leaving the result in the register.

and - Bitwise and

1111	dddd₄	1000	dddd₄

Flags: Z,N

Performs the bitwise AND of two data registers, leaving the result in the destination register.

brCC - Conditional branch

0000	0000	0001	cccc
sxxxxxxxxxxxxxxx			

Flags: None

If the condition is true then (offset $<<$ K_{isa}) is added to nipar.

bru - Unconditional branch

0000	0000	0001	0000
sxxxxxxxxxxxxxxx			

Flags: None

(offset $<<$ K_{isa}) is added to nipar.

byte - Byte align

```
1101 | dddd₁ | 1111 | aaaa₂
```

Flags: Z,N

Conditionally shift right 8 bits and mask. Used after a load instruction to align 8 bit data read from word offsets. If the address contained in the source address register is on an 8 bit boundary (has bit 2 set), then the value in the data register is shifted right 8 bits. If the address is not on an 8 bit boundary, then the upper 8 bits of the register are cleared.

NOTE: The negative flag is set with bit 7, not bit 15. This facilitates sign extension of 8 bit quantities.

cmp - Compare data registers

```
1110 | dddd₁ | 1000 | dddd₂
```

Flags: Z,N,V,C

Sets flags for magnitude comparison of two data registers by subtracting the source register from the destination register, affecting only the flags.

div - Signed 32 by 16 division

```
1110 | dddd₁ | 0101 | dddd₂
```

Flags: Z,N,V,C

Signed division of a 32 bit signed integer by a 16-bit signed integer, returning the 16-bit signed quotient and remainder. The 32-bit dividend is stored (little-endian) in two consecutive registers starting from the index of the destination register. The 16-bit divisor is in the source register. The remainder is returned in the destination register, and the quotient is returned in the register after the destination register (modulo 16). An overflow occurs if the quotient requires more than 16-bits to represent.

eadd - Add data register to address register

```
0101 | aaaa₁ | 0010 | dddd₂
```

Flags: Z,N,V,C

Adds a data register to an address register, leaving the result in the address register.

eaddq - Add quick constant to address register

```
| 1000 | aaaa₄ | sxxxxxxx |
```

Flags: Z,N,V,C

Adds an 8 bit signed constant to an address register, leaving the result in the address register.

ecmp - Compare of address registers

```
| 0101 | aaaa₄ | 0100 | aaaa₃ |
```

Flags: Z,N,V,C

Sets flags for magnitude comparison of two address registers by subtracting the source register from the destination register, affecting only the flags.

efadd - Add address register to address register

```
| 0101 | aaaa₄ | 0110 | aaaa₃ |
```

Flags: Z,N,V,C

Adds two address registers, leaving the result in the destination register.

efsub - Subtract address register from address register

```
| 0101 | aaaa₄ | 0111 | aaaa₃ |
```

Flags: Z,N,V,C

Subtracts the source register from the destination register, storing the result in the destination register.

eldf - Extended load forward

```
| 0110 | aaaa₄ | 1101 | aaaa₃ |
```

Flags: Z,N

Post-increment load into address register. Memory is read from address pointed to by the source register, and placed into the destination register. The source register is then incremented.
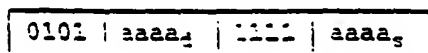
eldi - Load immediate into address register

1110	aaaa_d	0000	1000
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
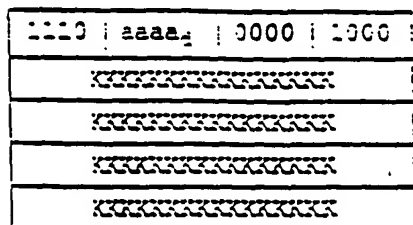
Flags: Z,N

Load 64-bit constant into an address register

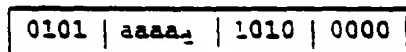# emov - Move address register

0101	aaaa_d	1111	aaaa_s

Flags: Z,N

Move the value from the source address register to the destination address register.

erotl - Rotate address register left through carry flag

0101	aaaa_d	1010	0000

Flags: Z,N,C

Shift an address register to the left one bit. The LSB is replaced with the value of the carry flag. The MSB is placed into the carry flag at the end of the instruction.

esl - Shift address register left

0101	aaaa_d	1000	0000
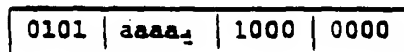
Flags: Z,N

Shift an address register to the left one bit.

esr - Shift address register right

```
| 0101 | aaaa₄ | 1001 | 0000 |
```

Flags: Z,N

Shift an address register to the right one bit.

est - Store address register

```
| 0110 | aaaa₄ | 1000 | aaaa₃ |
```

Flags: Z,N

Store from an address register. The 64-bit value in the source register is written to the memory location pointed to by the destination register. The value is written as four 16-bit words placed in little-endian order.

estr - Extended store reverse

```
| 0110 | aaaa₄ | 1110 | aaaa₃ |
```

Flags: Z,N

Pre-decrement store from address register. The destination register is decremented, and then the value in the source register is written to the memory location pointed to by the destination register. The value is written as four 16-bit words placed in little-endian order.

esub - Subtract data register from address register

```
| 1110 | aaaa₄ | 0010 | dddd₃ |
```

Flags: Z,N,V,C

Subtracts a data register from an address register, leaving the result in the address register.

inv - Bitwise inverse

```
| 1111 | dddd₄ | 0101 | dddd₃ |
```

Flags: Z,N

Places the bitwise inverse of the source register into the destination register.

---

jCC - Conditional jump

0000	0000	0000	cccc
xxxxxxxxxxxxxxxx			
xxxxxxxxxxxxxxxx			
xxxxxxxxxxxxxxxx			
xxxxxxxxxxxxxxxx			

Flags: None

Conditional jump to absolute address. See Table 7 on page 22 for condition code bit definitions.

jmp - Unconditional jump

0000	0000	0000	0000
xxxxxxxxxxxxxxxx			
xxxxxxxxxxxxxxxx			
xxxxxxxxxxxxxxxx			
xxxxxxxxxxxxxxxx			

Flags: None

Unconditional jump to absolute address. Same as jCC with condition "always".

jsr - Jump to subroutine

0001	aaaa$_d$	0000	aaaa$_s$

Flags: None

The destination register is first incremented, then the current nipar (pointing to the next instruction) is stored to the address pointed to by destination register (usually the stack pointer). nipar is then loaded with the address in the source register prior to fetching the next instruction.

ksl - Shift left by constant

1101	dddd$_d$	1000	KKKK

Flags: Z,N

Shift a data register to the left by a constant number of bits.

---

ksr - Shift right by constant

```
| 1101 | dddd₄ | 1001 | KKKK |
```

Flags: Z,N

Shift a data register to the right by a constant number of bits.

ld - Load data register

```
| 0110 | dddd₄ | 0001 | aaaa₅ |
```

Flags: Z,N

Load a data register from memory. The value pointed to by the source address register is loaded into the destination data register.

ldf - Load forward

```
| 0110 | dddd₄ | 0101 | aaaa₅ |
```

Flags: Z,N

Post-increment load into data register. Memory is read from address pointed to by the source address register. and placed into the destination data register. The source register is then incremented.

ldi - Load immediate

```
| 0111 | dddd₄ | 0000 | 0000 |
| KKKKKKKKKKKKKKKK |
```

Flags: Z,N

Load a 16-bit immediate value into a data register

mask - Bitwise mask operation

```
| 1111 | dddd₄ | 0100 | dddd₅ |
```

Flags: Z,N

The destination register is replaced with the bitwise inverse of the source register anded with the destination register.

mov - Move data register

```
| 1111 | dddd₄ | 1010 | dddd₅ |
```

Flags: Z,N

The value in the source data register is placed into the destination data register.

mul - Signed 16 by 16-bit multiply

```
| 1110 | dddd₄ | 0100 | dddd₅ |
```

Flags: Z,N,V,C

The result of multiplying the value in the source register by the value in the destination register is stored (little endian) in the two consecutive registers starting with the destination register.

or - Bitwise or

```
| 1111 | dddd₄ | 1110 | dddd₅ |
```

Flags: Z,N

Performs the bitwise OR of two data registers, leaving the result in the destination register.

rotl - Rotate data register left through carry flag

```
| 1101 | dddd₄ | 1010 | 0000 |
```

Flags: Z,N,C

Shift a data register to the left one bit. The LSB is replaced with the value of the carry flag. The original MSB is placed into the carry flag at the end of the instruction.

rti - Return from interrupt.

```
| 0001 | aaaa₅ | 1001 | 0000 |
```

Flags: Z,N,V,C

See Section 1.4 on page 3. The source register is used as the stack pointer.

rts - Return from subroutine

0001	aaaa₁	0001	0000

Flags: None

Return from subroutine. nipar is loaded from the memory location pointed to by the destination register (usually the stack pointer). The destination register is then incremented.

sl - Shift left

1101	dddd₁	0000	dddd₂

Flags: Z,N

The destination register is shifted left by the number of bits specified by the value in the source register.

sr - Shift right

1101	dddd₁	0001	dddd₂

Flags: Z,N

The destination register is shifted right by the number of bits specified by the value in the source register.

st - Store

0110	dddd₁	0000	aaaa₂

Flags: Z,N

Store from a data register. The value in the source register is written to the memory location pointed to by the destination register.

str - Store reverse

```
| 0110 | dddd₄ | 0110 | aaaa₃ |
```

Flags: Z,N

Pre-decrement store from data register. The destination register is decremented, and then the value in the source register is written to the memory location pointed to by the destination register.

sub - Subtract data register from data register

```
| 1110 | dddd₄ | 0001 | dddd₃ |
```

Flags: Z,N,V,C

Subtracts the source register from the destination register, storing the result in the destination register.

subc - Subtract with carry

```
| 1110 | dddd₄ | 0011 | dddd₃ |
```

Flags: Z,N,V,C

Subtracts the source register from the destination register, then subtracts the carry bit, storing the result in the destination register.

trap - Unconditional trap

```
| 0011 | aaaa₄ | 0000 | 0000 |
```

Flags: None

Execute interrupt handler. See Section 1.4 on page 3. The destination register is used as the stack pointer.

udiv - Unsigned 32 by 16-bit division

```
| 1110 | dddd₄ | 0111 | dddd₃ |
```

Flags: Z,N,V,C

Unsigned division of a 32 bit signed integer by a 16-bit signed integer, returning the 16-bit signed quotient and remainder. The 32 bits are stored (little-endian) in two consecutive registers starting from the index of the destination register. The divisor is in the source register. The remainder is returned in the destination register, and the quotient is returned in next register after the destination register. An overflow occurs if the quotient requires more than 16-bits to represent.

umul - Unsigned 16 by 16-bit multiplication

```
| 1110 | dddd₄ | 0110 | dddd₃ |
```

Flags: Z,N,V,C

The result of multiplying the value in the source register by the value in the destination register is stored (little endian) in the two consecutive registers starting with the destination register.

xad - Transfer address register to data registers

```
| 0101 | dddd₄ | 1101 | aaaa₃ |
```

Flags: None

Transfer the value in the source address register to four consecutive data registers starting with the destination register. The value is stored little endian, and the destination register address is calculated modulo 16 so that the destination register may be any register.

xda - Transfer data registers to address register

```
| 0101 | aaaa₄ | 1100 | dddd₃ |
```

Flags: None

Transfer a little endian 64-bit value in four consecutive data registers into the destination address register. The source register address is calculated modulo 16 so that the destination register may be any register.

xor - Bitwise exclusive or

```
| 1111 | dddd₄ | 0110 | dddd₈ |
```

Flags: Z,N

Performs the bitwise exclusive OR of two data registers, leaving the result in the destination register.

xpcw - Exchange processor control word

```
| 0100 | dddd₄ | 1010 | 0000 |
```

Flags: All

The value in the source data register is exchanged with the pcw register.

xivec - Exchange interrupt vector

```
| 0101 | aaaa₄ | 0111 | 0000 |
```

Flags: All

The value in the source address register is exchanged with the ivec register.

4.0 Condition codes

The condition code opcode subfields use the values from the following table:

TABLE 7. Condition Codes

Mne.	Code	Meaning	Equation
eq	0001	equal	Z
ne	0010	not equal	$\sim Z$
gt	0011	greater than	$Z \sim (N \oplus V)$
lt	0100	less than	$N \oplus V$
ge	0101	greater or equal	$\sim (N \oplus V)$
le	0110	less or equal	$Z \mid (N \oplus V)$
p	0111	positive	$\sim N$
n	1000	negative	N
gtu	1001	greater than, unsigned	$\sim C \sim Z$
vc	1010	overflow clear	$\sim V$
vs	1011	overflow set	V
cc	1100	carry clear	$\sim C$
cs	1101	carry set	C

APPENDIX B

5

INSTRUCTION SET 1,
A PIPELINED MULTIPLY-ACCUMULATE ISA

ISA1 - Pipelined Convolution Engine for XC4013

Introduction

ISA1 is a pipelined multiplier-accumulator array capable of 4 simultaneous multiply-accumulates per instruction cycle. There are 8 8-bit data register (xd0-xd3 & yd0-yd3), one for each input to the four 8-bit X 8-bit multipliers. The four multiplier outputs are summed together via a pipelined adder array, until one final 16-bit sum emerges, where up to four 16-bit registers may store the result (m0-m4). The architecture of ISA1 presumes a flow-through batch processing cycle with main memory. As such there is no feedback path through the multiplier accumulator data path for recycling accumulated results, because the emphasis is on memory data flow rates. There is no provision for overflow scaling or for extended finitude accumulates; ISA1 presumes that the coefficients used for convolutional filtering yield not more than 16-bit result finitude for all data sets. The multiplication array assumes 8-bit 2's complement data inputs, and produces a 16-bit 2's complement result.

Access to memory is managed by two 64-bit address registers (a0 & a1), which may be thought of as interchangeable source and destination pointers. Program flow is managed by the standard 64-bit NIPAR register, and a 64-bit interrupt vector register is supported (IVEC) for a lone interrupt, such as frame or data-ready interrupt.

The instruction set of ISA1 is very small, and aligned to 16-bit word size, matching the $K_{ISA}=4$ memory organization for the general purpose outer-loop processor ISA0. Up to 7 arithmetic operations may be instantiated in a single clock cycle with ISA1, and the implementation sustains result rate of one per clock over small windows of clocks, with the ability to index new source or destination addresses, and move register data from and to memory, in parallel with computation.

ISA1 Instruction Set

Data Movement

ld (reg-vector)   Any of up to 14 registers are loaded sequentially from memory, according to 14-bit bitmap reg-vector contained right-justified in the instruction word.

st (reg-vector)   Any of up to 14 registers are stored sequentially into memory, according to 14-bit bitmap reg-vector contained right-justified in the instruction word.

ld (ivec-data)   The 64-bit address following this instruction is loaded into the IVEC register, while NIPAR+=5 to point to the next instruction to execute.

Program Control

jmp (nipar-data)   The 64-bit address following this instruction is loaded into the NIPAR register, thereby pointing to the next instruction to execute.

ISA1 Instruction Set, con't...
Arithmetic
mac (m-reg)   The multiplication-result register indicated by 2-bit m-reg code receive the product and sum $(xd0*yd0)+(xd1*yd1)+(xd2*yd2)+(xd3*yd3)$.

macp (s-vec, d-vec)   The multiplication-result register indicated by 2-bits of the 4-bit d-vec code receive the product and sum $(xd0*yd0)+(xd1*yd1)+(xd2*yd2)+(xd3*yd3)$. One other bit of the d-vec code selectively enables a memory write of this result register at address (a1), while the remaining bit of the d-vec code selects whether address register a1 is incremented or not. The 8-bit s-vec is divided into four 2-bit groups which specify successively for data registers xd0-xd3 whether, a read from memory at address (a0) is to occur, and whether address register a0 is to be incremented. If reads or writes are specified, they are performed in parallel with multiplication. Software must account for the pipelined alignment of instruction processing for batches of data read from and stored to memory.

Reconfiguration
reconf(ISA-vector)   ISA1 is de-contexted, and the S-machine is reconfigured for the ISA selected by the ISA-vector bitfield in the instruction.

Block Diagram of ISA1 - Pipelined Convolution Engine for XC4013